University of Alberta

THE COMPETENCY REFINERY: AN ENVIRONMENT FOR PROCESS IMPROVEMENT THROUGH THE ACCUMULATION OF EXPERIENCES

by

Amr Atef Kamel ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Doctor of Philosophy**.

Department of Computing Science

Edmonton, Alberta
Spring 2003

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisisitons et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 0-612-82123-4
Our file Notre référence
ISBN: 0-612-82123-4

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou aturement reproduits sans son autorisation.

Canada

University of Alberta

Library Release Form

Name of Author: Amr Atef Kamel

Title of Thesis: The Competency Refinery: an environment for process improvement through the accumulation of experiences

Degree: Doctor of Philosophy

Year this Degree Granted: 2003

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as herein before provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

Amr Atef Kamel
611B Michener Park
Edmonton, Alberta
Canada, T6H 5A1

Date: Apr. 14, 03

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **The Competency Refinery: an environment for process improvement through the accumulation of experiences.** submitted by Amr Atef Kamel in partial fulfillment of the requirements for the degree of **Doctor of Philosophy**.

Paul G. Sorenson (supervisor)

H. James Hoover

Eleni Stroulia

Petr Musílek

Hausi Müller (external examiner)

Date: Apr 8, 03

Abstract

Accumulating and managing development experiences plays a key role in improving software quality and process. However, the complexity of the software process makes it difficult to establish and effectively provide operational support for experience management. To overcome these difficulties, this thesis defines a concept named the Competency Refinery, along with a method for building and running one to support software process improvement. The Competency-Refinery concept provides an organizational approach for extracting development experiences from current software projects and supplying them to future projects. The thesis also provides a set of tools to support the accusation, selection, and evolution of software experiences within the Competency Refinery framework.

The concepts defined in this thesis, has been deploying to implement a Competency Refinery to support software development using application frameworks. Then, the concepts were validated by using the refinery, over two years, to support software development in 15 different software projects developed as a part of a senior level software engineering course at the Department of Computing Science, University of Alberta. Two experience-bases were developed and managed within the framework of this thesis. One to support peer reviews and the other to support development using a specific framework called the CSF. Through our experience with the competency refinery, a peer review process for information exchange was identified to support framework learning.

Acknowledgements

All praises and thanks are due to GOD who guides and protects me. Indeed his bounties on me are countless. The work done in this thesis and in the years to come is devoted for his sake.

This thesis would have been impossible without the help and support of many people. First and foremost, my supervisor Dr. Paul G. Sorenson, who supported me technically, financially and morally; his patient encouragement through out the thesis is very much appreciated. Thanks also must be extended to members of my supervisory committee, Dr. H. James Hoover and Dr. Eleni Stroulia and other members of my examining committee Dr. Petr Musílek and Dr. Hausi Müller for reading my thesis and for their individual comments and suggestions. Special thanks are due to Dr. Lettice Tse and Dr. Garry Froehlich for their help and participation in the case study.

A large dept of gratitude is also owed to many of my friends who encouraged me when I needed the most, and supported my in my down time. I would like to specially thank Khaled Obaia and Omar El-sherbini; without their support and encouragement, I would never have been where I am today. I also would like to express my sincere gratitude to my parents and my in-laws for all their love, support and encouragement. Last but definitely not least, I am eternally grateful to my wife Solafa, her unconditional love and many sacrifices for my sake gave me the courage to get over my down times and the will to finish this thesis.

Contents

1 Introduction 1
- 1.1 Quality improvement initiatives . . . 2
- 1.2 Research problem and method . . . 3
  - 1.2.1 Research Method . . . 4
- 1.3 Thesis contributions . . . 6
- 1.4 Thesis outline . . . 7

2 Learning Software Organizations 8
- 2.1 Introduction . . . 8
- 2.2 Software engineering knowledge . . . 9
  - 2.2.1 Knowledge representation . . . 10
- 2.3 Knowledge management . . . 12
  - 2.3.1 Knowledge evolution cycle . . . 13
  - 2.3.2 Strategies for knowledge management . . . 14
- 2.4 Organizational structure . . . 15
  - 2.4.1 Examples of knowledge units . . . 18
- 2.5 The experience factory paradigm . . . 19
  - 2.5.1 Discussion . . . 21
- 2.6 Technologies to support the experience factory . . . 22
  - 2.6.1 Knowledge acquisition . . . 23
  - 2.6.2 Knowledge deployment . . . 24
  - 2.6.3 Knowledge creation and organization . . . 25
- 2.7 Implementation of the experience factory concept . . . 26
- 2.8 Summary . . . 28

3 A Model for Selecting Process Steps 30
- 3.1 Introduction . . . 30
- 3.2 The Competency Refinery concepts . . . 31
- 3.3 Competency Refinery architecture . . . 33
  - 3.3.1 Reference architecture for the Competency Refinery . . . 34
  - 3.3.2 Instantiation of the architecture . . . 36
- 3.4 Packaging software process knowledge . . . 38
  - 3.4.1 Types of process packages . . . 39
  - 3.4.2 Experience Representation . . . 42
  - 3.4.3 Selection criteria . . . 43

3.4.4 Experience Acquisition 44
3.5 Summary 45

4 An Experience Base for Peer Reviews 47
4.1 Introduction 47
4.2 Related work on taxonomies for peer reviews 48
4.3 Background on peer reviews 50
4.3.1 Empirical Studies 50
4.3.2 Experience Reports 51
4.4 A framework for process taxonomy 53
4.5 A taxonomy for peer reviews 55
4.5.1 Technical dimension 55
4.5.2 Economic dimension 61
4.5.3 Support Dimension 63
4.6 Summary 68

5 Enacting the Competency Refinery 69
5.1 Introduction 69
5.1.1 Study objectives 70
5.2 Background 71
5.3 Refinery context 73
5.3.1 Study participants 74
5.3.2 Organizational Structure 74
5.3.3 Development process 75
5.4 Projects context 76
5.4.1 The CSF framework 77
5.4.2 Peer review for information sharing 78
5.5 Data and analysis 80
5.5.1 Data and analysis technique 80
5.5.2 Potential confounding factors 82
5.6 Results 85
5.6.1 Effectiveness of peer reviews 85
5.6.2 Role of the documentation 86
5.6.3 Effective processes 89
5.7 Summary 90

6 Packaging Process Experiences 92
6.1 Introduction 92
6.2 Investigation strategy 93
6.2.1 Evaluation criteria 94
6.3 Overview of data 94
6.3.1 Data reduction 95
6.3.2 Summary of observations 96
6.4 The effect of process structure 98
6.4.1 The effect of the detection technique 98

6.4.2 Large team versus small team 99
6.4.3 One session versus multiple sessions 102
6.5 The effect of the process inputs 103
6.5.1 Professional training factor 104
6.5.2 Industrial experience factor 106
6.5.3 Preparation time factor 107
6.6 Summary and recommendations 108
6.6.1 Summary 111

7 Automated Support for the Competency Refinery 112
7.1 Introduction 112
7.2 Requirements for supporting the Competency Refinery 113
7.2.1 Functional requirements. 113
7.2.2 System level requirements 114
7.3 A prototype environment to support the Competency Refinery 117
7.3.1 Supporting Technology 118
7.3.2 Tool Architecture 118
7.4 System implementation 120
7.4.1 Current status of the experience base 124
7.5 Usage scenarios 125
7.5.1 Selecting an experience package 125
7.5.2 Add a new feature 126
7.6 Assessment and proposed modifications 127
7.7 Summary 131

8 Conclusions and future work 133
8.1 Summary 133
8.2 Contributions and results 134
8.2.1 Building and running a Competency Refinery 135
8.2.2 Packaging process experiences 135
8.2.3 Automated support for the Competency Refinery . . . 136
8.2.4 Process taxonomy model for peer reviews 137
8.2.5 Documentation of a major case study 137
8.2.6 Supporting framework knowledge internalization 138
8.3 Future directions 138
8.3.1 Tool development and refinement 138
8.3.2 Extending the experience base 139
8.3.3 Case studies 140
8.3.4 Controlled experiments 140
8.3.5 Documenting application frameworks 140
8.3.6 Peer reviews for framework understanding 141
8.3.7 Forecasting and estimation 141
8.4 Concluding remarks 142

Bibliography 144

A Case Based Reasoning 161
A.1 The CBR process 161
A.2 Building a CBR System 163
A.2.1 The case base 163
A.2.2 Indexing techniques 164
A.2.3 Retrieval algorithms 164
A.2.4 Adaptation strategy 164

B Modelling Current Peer Reviews 165
B.1 Inspection 165
B.1.1 Fagan Inspection 165
B.1.2 Fine-tunes on Fagen Inspection 167
B.1.3 Gilb Inspection 167
B.1.4 Phased Inspections 168
B.1.5 Inspecting for Program Correctness 169
B.2 Technical Review 169
B.2.1 Round-Robin Review 169
B.2.2 Active Design Review 170
B.2.3 Verification Based Reviews 171
B.2.4 Selected Aspect Review 171
B.2.5 Meeting-less reviews 172
B.3 Walkthrough 172
B.3.1 Structured Walkthrough 173
B.3.2 Technical Walkthrough 173
B.3.3 Freedman and Weinberg's Walkthrough 174
B.3.4 Cognitive Walkthrough 175
B.3.5 Programming Walkthrough 175

C Support Material for the Case Study 177
C.1 Post-review questionnaire 178
C.2 Post-project questionnaire 179
C.3 Scenario-based checklist 181
C.4 Forms 182

List of Figures

2.1 Knowledge evolution cycle 13
2.2 Differences in knowledge units deployment 16
2.3 Conceptual structure of the experience factory 20

3.1 The Competency Refinery 32
3.2 Relation between the competency refinery and the knowledge evolution cycle 37
3.3 Different levels of experience packages 41
3.4 Experience package template 44
3.5 Sources of knowledge for different types of packages 45

4.1 Dimensions and attributes of proposed taxonomy of peer reviews 56

5.1 Percentage of students relying on the framework experience base to understand the CSF 88

6.1 Disposition of issues recorded at the review meetings 96
6.2 Number of findings per reviewer during preparation 97
6.3 Time spent preparing for the review 98
6.4 The effect of using different checklists 99
6.5 Review benefits as team size increases 101
6.6 Difference in review benefits as team size increases 102
6.7 The effect of multiple session on review results 104
6.8 Histogram of randomly selected academic information 105
6.9 Histogram of consistent industrial data through random selection 107
6.10 Histogram of randomly selected effort data 108
6.11 Findings breakdown 109

7.1 Decomposition of the experience administration tasks 114
7.2 Architecture of the competency refinery support environment . 119
7.3 Maintenance tool screen 122
7.4 Query the RDBP experience base 126
7.5 Case description for Fagan inspection 127
7.6 Add a new question 128
7.7 Associate questions with cases 129

A.1 The process of case based reasoning 162

List of Tables

3.1 Activities and communication pathes for the competency refinery 34

4.1 Empirical results of peer review experiments 51
4.2 Reported peer review benefits in industry 53

5.1 Background scores for project teams 84
5.2 Correlation between implementation score and student background 84
5.3 Answers to question: "how helpful were the reviews in understanding the CSF?" 86
5.4 Rating for different sources of knowledge 87
5.5 Background scores for project teams 90

6.1 Multiple session - data summary 103
6.2 Data summary for industrial experience versus reported findings 106

7.1 Assessment of the prototype 129

A.1 An example of a case 163

Chapter 1

Introduction

The need to manage the quality of software products in a better way is quite evident. Spectacular failures such as the crash of the AT&T communication network [15] and Ariane 5 launch [217] still occur. Many recent problems, from loss of money [55] to endangering human life [201], can be attributed to quality problems in software products [124]. In addition to quality problems, schedule and budget overruns are commonplace. For example, a computer system for Allstate Insurance was budgeted for $8 million and scheduled to complete in 5 years; the system was completed after 11 years of development with a total cost of $100 millions [194]. Unfortunately, defects are bound to occur even in the most carefully written software.

Quality management has remained more of an art than a science. Most project managers depend on their experiences and rather ill defined heuristics to manage software production. With this approach, software managers are expected to mentally maintain and deploy processes that produce high quality software in their particular development setup. Unfortunately, the success of this approach is tightly related to the level of experience of the project manger.

Striving to rectify these problems, many researchers turned their attention towards the software development process [162]. After decades of researching the software process, many practices have been identified and proven to be useful in improving the quality of work products. However, the complexity of most processes makes it difficult for organizations to identify and assess the parameters affecting the process. As a result, selecting the proper process, from available process alternatives, remains a challenge.

This dissertation addresses the above challenge by capitalizing on experiences gained during software development. The approach is based on extracting and packaging development experiences into an experience base, and making it available for software practitioners. Software practitioners make use of the packaged knowledge by extracting experience packages from the experience base, and then reusing or tailoring them to suite the particulars of their development environment. Practitioners can also enrich the experience base by including their experiences to the experience base.

The rest of the chapter discusses the trends in quality improvement initiatives, states the research problem addressed in this thesis along with the research methodology followed. A synopsis of the thesis contributions is presented and finally, the chapter ends with an outline for the rest of the thesis.

1.1 Quality improvement initiatives

Lack of a reliable methodology and the highly dynamic software market have lead to instituting quality improvement programs in many software organizations. The majority of these programs are based on the concept of continuous process improvement. The concept promotes the idea of achieving quality improvement in small steps by establishing feedback loops to monitor the performance of the improvement efforts.

Many quality improvement programs have been tried in the last two decades. Some methods are based on the Shewart-Deming Cycle (plan/do/check/act) [65], others used the closely related Total Quality Management (TQM) paradigm [79]. However, these approaches suffer from two major problems [21]: they assume a consistent picture of a good software product and, in general, they don't deal specifically with the dynamic, evolutionary, nature of software development.

To overcome these problems, Basili [21] introduced the Quality Improvement Paradigm (QIP). The paradigm is based on the notion that improving software process and product requires a continuous learning through accumulating experiences in well-understood forms and models that can be accessible to other projects for use and/or modification. Basili *et al.* [24] introduced the experience factory concept to support the enaction of the quality improvement paradigm. The experience factory institutionalizes continuous improvement through the capture and utilization of the collective learning of the organization. Through the experience factory mechanisms different experiences are collected and analyzed then packaged in order to provide, upon request, feedback to new projects based upon the experiences of similar projects.

The perceived benefits of the experience factory paradigm were supported by the Software Engineering Laboratory (SEL) success story [23]. Over a ten-year period, the experience factory at SEL managed to reduce development cost by 60%, decrease the error rate by 85%, and reduce cycle time by 20%. Currently, the experience factory concept is widely accepted by the software industry as the process management approach that is most suitable for the special needs of software development [226].

1.2 Research problem and method

In spite of reported successes, quality improvement programs are not always successful; even when deploying practices, such as the experience factory, that worked well for other organizations. The limited success can be attributed to a combination of factors [208]:

- concentrating on the mechanics of the technology rather than acting on the information it provides,
- enacting expensive practices that are beyond the technical needs of the process, and
- overlooking the organizational culture while defining the improvement program.

Furthermore, building an experience factory still represents a major challenge that is undertaken by very few software organizations [106]. It is an extremely challenging undertaking for the following reasons:

- The concept has been treated theoretically; however, reported efforts in building an experience factory indicate that available information is too abstract to help implement it.

- The experience factory at SEL evolved over more than 15 years. The establishment process was written with hindsight, consequently the details of the process may have been lost over time.

- Little information was delivered about how experiences can be accumulated, formalized, stored and used in day-to-day process management.

The goal of this thesis is to specify, implement and evaluate a model for process improvement based on the experience factory concept. The goal can be refined to the need to build a model for selecting processes based on the accumulation of experiences. The general questions we would like to address in this research area can be defined as follows:

> How can we build an experience base to capture experiences gained during project development? and what are the proper methods to populate and extract information from the experience base?

Clearly, these questions are much more too broad to be fully addressed in one thesis. The work in this thesis begins a long-term exploration by focusing on some of the fundamental issues in this area. Specifically, we developed the concept of a Competency Refinery that defines how experiences can be captured, analyzed, modelled and deployed in support of software development in the *focused* domain of application frameworks.

1.2.1 Research Method

Based on the type of the phenomena under investigation, March & Smith [153] distinguished between design and natural sciences. In general, they divided research activities into: build/evaluate for design science and theorize/justify for natural science. They argued that the former is suitable for artificial phenomena whereas the latter is suitable for natural phenomena. They further categorized outputs produced by design research into: representational constructs, models, methods, and instantiations. Since we are investigating the building of a Competency Refinery, the proposed research falls into the design science category.

Nunamaker *et al.* [173] presented a methodology for design science. The methodology consists of concept building, system development, experimentation and observation. During concept development, new ideas are explored and conceptual framework for methods and models are constructed. The system development phase is dedicated to build concrete systems to realize the conceptual frameworks. Finally, during experimentation and observations research methods such as action research, laboratory experiments, field tests, simulations and experimentation aid the researchers in validating or rejecting the concepts upon which the system was built.

Zelkowitz & Wallace [244], defined three different experimentation models for validating technology: observational, historical and controlled. Observational methods collect relevant data as projects develop. They rely on unobtrusive research methods such as project monitoring, case studies, field studies and assertions. Historical methods collect data from projects that have already been completed. Since data already exists in historical methods, the focus is primarily on analyzing data using techniques such as literature search or postmortem analysis. Controlled methods provides multiple instances of an observation for statistical analysis. This method is the classical method for experimental design in other scientific disciplines; relying on methods such as replicated experiments in laboratory setting and simulations.

In his Turing Award Lecture, Hartmanis [101] endorsed demonstration and observation as validation techniques in computer science, as they suite the nature of the computer science research. In this research, model evaluation was run as a case study. The building and execution of a Competency Refinery was monitored in order to collect information about various attributes characterizing the experiences.

Specifically, two types of experience bases are managed within the Competency Refinery defined in this thesis. One to support peer reviews and the other to support development using a specific framework called the CSF (Client Server Framework) [85]. The experience bases, and processes to interact with them are evaluated by monitoring teams of developers using the CSF framework to assess the influence of the experience bases on the development process.

1.3 Thesis contributions

The work presented in this thesis is intended to address the above mentioned deficiencies in the experience factory. In particular, it makes the following contributions:

- The development of a method for building and running a Competency Refinery to support software development. The applicability of the method is examined by providing a concrete implementation for the refinery.
- A three-level model for packaging process experiences. The model supports the documentation of hands-on experiences as well as more abstract level of experiences such as lessons learned.
- The development of a three dimensional process taxonomy model. The objective of this model is to support engineering decision making in the software process domain. The taxonomy was used to develop a process model for peer reviews, thereby providing the refinery agents, and the software practitioners at large, with a frame of reference for comparing different aspects of the review process.
- The provision of an integrated set of tools that support knowledge management within the refinery context. The set of tools support the selection, evolution and acquisition of experiences within the competency refinery.
- The documentation of a significant case study that demonstrates the applicability of the approach in the development of applications using a

common framework.

1.4 Thesis outline

The body of this thesis starts, in Chapter 2 with reviewing different approaches used for building a learning software organization. Work done to implement experience factories is also discussed. Chapter 3 provides the specifics of the experience management environment used in this thesis research and presents the characteristics of the structure agents, details of the experience base, as well as the methodology developed to build and run the structure. The proposed process taxonomy is presented in Chapter 4. Efforts to identify experience packages and build the experience base for peer reviews are also reported in this chapter.

Details of the case study are discussed in Chapters 5 and 6. Enaction details are reported in Chapter 5, and the evolution of the experience package are discussed in Chapter 6. Chapter 6 also presents a simulation experiment that illustrates how performance prediction capabilities can be added to the experiences extracted in the case study. Chapter 7 discusses tool support for the refinery and introduces a prototype tool intended to support the experience base as well as tools for collecting and analyzing information. Finally, thesis conclusions and further work related to this research are presented in Chapter 8.

Chapter 2

Learning Software Organizations

2.1 Introduction

The continuous change in technology, unpredictable strategies of competitors and the rapid change in customer needs has lead to a greater emphasize on knowledge as an important asset in the software industry [13]. To address these challenges, while providing high quality products, organizations are: systematically and continuously, managing their experiences for comprehensive reuse [5], leveraging the knowledge of highly skilled and experienced employees [80] and/or extending their knowledge from sources external to the organization [195].

Organizations taking measures to improve their products and processes by creating new knowledge and disseminating this knowledge through the organization are sometimes called *Learning Software Organizations* (LSO) [195]. Typically, LSOs have to deal with two fundamental issues: *knowledge management* and *organizational infrastructure* to support it. However, creating a LSO is not only a technical issue, it usually involves cultural change in the organizations. A culture that promotes continuous creation of knowledge and fosters the exchange of experience must be established.

Published studies [23], [106], [44], [116], [63] indicate a significant difference in knowledge management activities among software companies. In many cases, the recommended practices are in conflict. Therefore, similarities and differences among these practices have to be well understood before making a decision about the practices that best fit a particular organization. Because of this uncertainty, some experts suggest that maintaining a combination of strategies is the only way to improve the organization's ability to compete in the market [152].

The rest of this chapter will survey the work done in LSOs in preparation for presenting the model used as the foundation for this thesis. Sections 2.2 and 2.3 present issues in knowledge management related to the software organizations. Similarities and differences among organizational structure proposed to support knowledge management are presented in Section 2.4. Section 2.5 discusses the experience factory paradigm. Technologies proposed to support the factory's implementation, followed by presentation of some concrete implementations are discussed in Sections 2.6 and 2.7 respectively.

2.2 Software engineering knowledge

There is a wide variety of expertise to capture for a software organization. This expertise falls along three dimensions [13]: domain, methodology and technical expertise. Domain expertise incorporates knowledge about the application domain, methodology expertise requires knowledge about development processes and principles, and technical expertise encompasses knowledge about the development technology including, for example, tools used in development.

Basili *et al.* [22] recommended packaging software engineering knowledge in the form of experience packages. Baselines (e.g., resources and defect rates), models (e.g., quality and process) and definitions (e.g., process and tools) are examples of useful information that can be included in an experience package. Basili [21] made some general suggestions about the contents of different experience packages. He proposed six classes of packages:

1. **Product.** The center element of a product package is a life-cycle product supported with information needed to reuse this product, (e.g., programs, architectures and designs).

2. **Process.** The center element of a process package is a life-cycle process, supported with information to enact it and lessons learned from previous enactions, (e.g., process models, methods).

3. **Relationship.** The center element of a relationship package is relationships among different software project characteristics, (e.g., cost and defect models, resource models).

4. **Data.** The center element of a data package is data collected from different software projects, (e.g., standard quality records).

5. **Management.** The center element of a management package is reference information for project management, (e.g., guidelines, project handbooks, decision support models).

6. **Tool.** The center element of a tool package is reference information to a development and/or analysis tool, (e.g., CASE environment).

The contents and internal structure of a knowledge package depends upon the type of the packaged experience as well as its intended use. The types vary from mainly hands-on experiences [102] to lessons learned [222]. Generally, the package structure is clustered around a central element that determines the nature of the package. There is no general agreement on the internal structure of any of the proposed experience packages. In this research, we are focused on process knowledge; one of the goals is to define a process pattern for structuring and storing process experiences. The pattern should be capable of representing hands-on experiences and support the evolution of these experiences to recommended practices and lessons learned.

2.2.1 Knowledge representation

A central technical aspect of knowledge management is the experience base (sometimes called the Organization Memory (OM)) [228]. Typically, experience bases store a combination of informal, semi-formal and formal knowledge. This combination poses challenges in how to organize and represent knowledge.

This section discusses the merits of current frameworks used for packing different kinds of software engineering experiences.

Informal reports are the first form of packaged experiences proposed and used by Basili *et al.* [25]. Reports may contain any combination of plain text, graphs, tables and figures; the report structure is left to the report author. Unfortunately, informal reports are very dependent on their author's views and perspectives and are prone to the risk of missing some relevant information (e.g., clear definition of the context in which the experience was enacted).

The project PERFECT, funded by the European community, defined structure for documenting process experiences. The suggested structure [177], [76] consists of: context description, process models and quality models. Context description contains a characterization of the environment and the project. The process models describe activities, methods and role definitions of the development process. The quality model represents the relationships among different factors affecting the process. This type of experience packages is limited to the software process [135].

The idea of extending software development experiences with formal characterizations was presented by Prieto-Diaz [187] and further refined by Ostertag [174]. Ostertag presented a formal language for documenting experiences, and introduced a similarity-based search mechanism to locate experiences in the experience base. However, to take advantage of this mechanism, characteristics of the requested experience have to be described using the same formal language. The need to have designers and developers describe experiences in a restricted, non-natural, language limits the applicability of this approach in practice.

A similar representation formalism, called REFSENO (REpresentation Formalism for Software ENgineering Ontologies) was presented by Tautz & Gresse von Wangenheim [223]. The goal of REFSENO is to formally define an ontology for software engineering. Defining an ontology is a long, time-consuming process. It requires a good understanding of the parameters affecting the development environment before documenting an experience. This can pose a limitation for many software organizations trying to start knowledge management efforts.

Houdek *et al.* [106] proposed an approach based on rearrangement and reprocessing of captured experiences into quality patterns. The main concept of the quality pattern is to document problem solution patterns, domain descriptions, explicit rationales and the pyramid thinking principle [166]. In this approach, only experiences that can be fitted as quality patterns can be captured and stored. Usually experiences directly extracted from development projects take different forms that may not necessarily fit into a quality pattern. Using this approach, incorporating experiences from the development projects to the experience base can sometimes be problematic or limited.

Generally, experience documentation techniques are concerned with documenting successful experiences. With the exception of the quality patterns approach, hands-on experiences are not documented within the same documentation framework, because hands-on experiences are not always successful. This puts 'real' experiences at risk of being lost. Furthermore, by neglecting experiences with limited to no success some important information, such as reasons of failure, may be lost and mistakes may be repeated. In our approach, experiences are stored in a structured-text format. Different types of experience packages are used to store different levels of experiences.

2.3 Knowledge management

Although there is no generally agreed upon definition of knowledge management, it is universally accepted that transfer of knowledge resides at the center of knowledge management [64]. The ultimate goal is to improve skills within the organization by providing software professionals with the expertise required to accomplish their tasks better.

Cook & Brown [61] distinguished two types of knowledge: explicit and tacit. Knowledge that can be captured in a manual is referred to as explicit. Tacit knowledge, on the other hand, includes the conventions and metaphors by which individuals work together and share ideas. In order to facilitate the transfer of knowledge, knowledge management activities should enable the

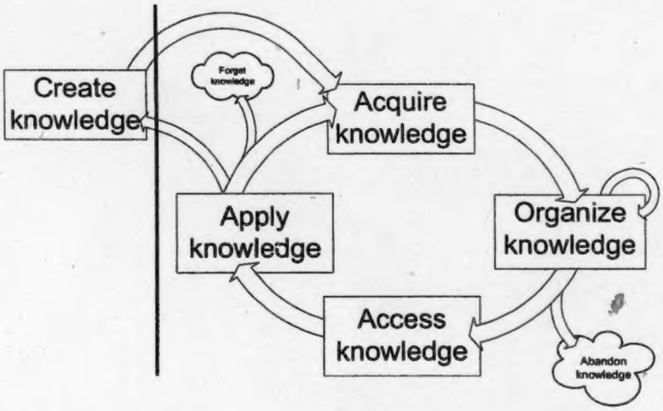


Figure 2.1: Knowledge evolution cycle

conversion of knowledge from tacit to explicit [152]. Captured knowledge can either be applied to similar tasks or linked in new ways to handle new tasks.

2.3.1 Knowledge evolution cycle

In a typical software organization, knowledge is created, operated upon during its lifetime, then forgotten or abandoned when it is no longer useful. This cycle, as shown in Figure 2.1, is usually referred to as the knowledge evolution cycle [196], [240].

The creation of knowledge in an organization is associated with formal training, innovation and creativity, or importation from outside sources. By recording human experiences, information about the knowledge is acquired, retained and preserved by the organization. The objective of the organizing stage is to process the captured information and present the knowledge in a format that is accessible and easy to use. Actions at these stages include, analyzing the information, classifying contents by attributes, providing indexes, assuring the quality of contents and providing access controls. During the access stage, knowledge is distributed to points of action. The distribution may take place using pull technologies, e.g. knowledge databases and search engines, or push technologies, e.g. training programs and alert mechanisms. Knowledge that does not get accessed is eventually abandoned from the evolution cycle. However, well understood knowledge may be abandoned as well, as it becomes common knowledge within the organization. Applying the knowledge is the ultimate goal and the most important stage of the cycle. Experiences from knowledge application contribute to the creation of new knowledge or the capture of more of its tacit part; hence deepening the knowledge understanding. Information from these experiences that are not acquired are usually forgotten.

2.3.2 Strategies for knowledge management

Trittmann [226] described two basic strategies for knowledge management: mechanistic and organic. The mechanistic form supports codified knowledge transfer. Systems adopting this form focus on capturing and documenting tacit knowledge. The organic form, on the other hand, supports personalized transfer of knowledge. Systems adopting this form focus on facilitating interpersonal communications.

The mechanistic form of knowledge management aims mainly at leveraging existing knowledge through knowledge packaging [226]; knowledge is package through codification and standardization. Systems aiming at knowledge leveraging supports the identification, documentation, storage and communication of the packaged knowledge. The packaging process requires a considerable upfront effort. However, once knowledge is documented, experience gained from one project can be distributed to several receivers with little effort. The benefits of knowledge leveraging is maximized if the packaged knowledge supports tasks with limited variability.

By combining knowledge from different sources, new knowledge can be created, and the innovation effect of knowledge transfer can be realized [47]. In these cases, targeted tasks are typically new, complex and poorly defined direct reuse of experience is unlikely. The interaction between these experiences is what fosters the innovation effects [172]. Systems targeting innovation should support the personalized forms of knowledge transfer in order to facilitate the exchange of ideas and the creation of permanent channels for feedback.

Knowledge management systems can target any identified area of expertise in the software industry: application domain, development methodology and/or technology. The objectives of a system and the type of tasks it targets determines the suitable form of knowledge management. For example, to support the core development processes, a mechanistic system would perform better than an organic one. On the other hand, the rapid rate of change in the technology makes the organic form a more suitable form of support for technology. However, there is growing belief [152] that maintaining a combination of both strategies is the right way to improve the organization's ability to compete in the market.

2.4 Organizational structure

In order to take advantage of the organizational knowledge, knowledge management activities should be supported by an organizational infrastructure dedicated to that purpose [12]. Published case studies [23], [106], [44], [116], [63] indicate a significant difference and, in some cases, conflict in knowledge management activities among software companies. In this section, we identify four points of variation (see Figure 2.2), that represent the main differences in organization structures. Examples of these structures are also discussed.

Tritmann [226] distinguished between two types of knowledge management structures: centralized and decentralized. In a centralized structure, (e.g., the experience factory [24]), an agent performs the same set of knowledge management tasks for all knowledge domains. This structure emphasizes capturing and representing knowledge in explicit formats. In a decentralized structure, (e.g., knowledge broker [44]), an agent performs all tasks pertaining to a certain knowledge domain. Maintaining lists of agents and their areas of experiences is the main focus of this structure [44]. The format of the preserved knowledge is left to individual agents to decide upon.

The relationship between the knowledge unit and the development organization represent another point of variation. The knowledge unit may serve as an internal consultant [64], or as supervisor that recommends how development should progress [204]. Knowledge domain and the level of experience

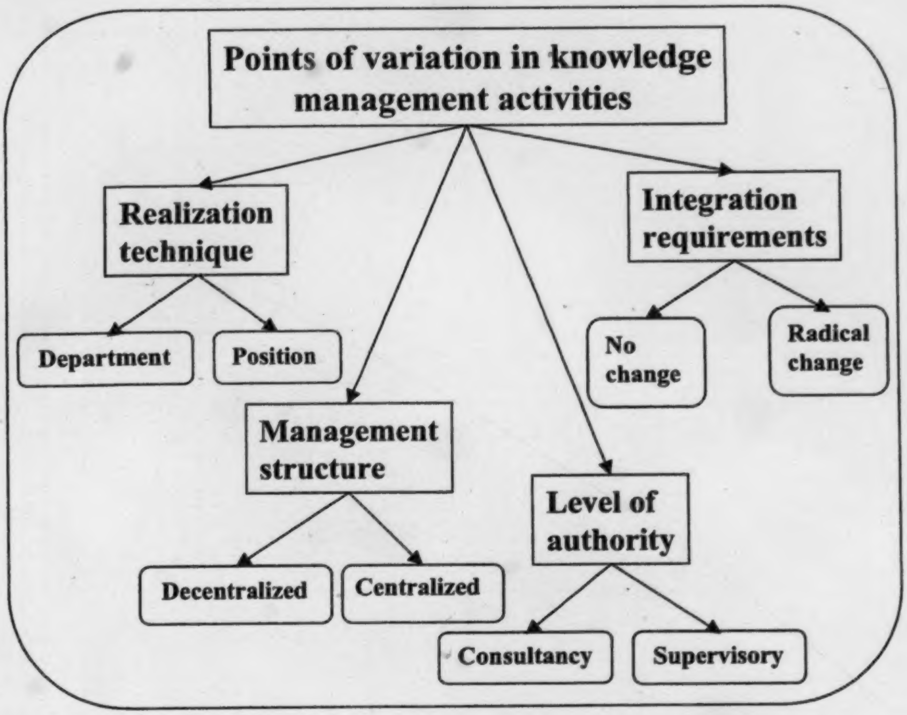

Points of variation in knowledge management activities
Realization technique
Department
Position
Management structure
Decentralized
Centralized
Level of authority
Consultancy
Supervisory
Integration requirements
No change
Radical change

Figure 2.2: Differences in knowledge units deployment

in the organization are the major factors in deciding the suitable type of relationship. Knowledge units dealing with methodology domains may adopt the supervisory or consultants relationship for various reasons. For example, a supervisory relation supports the injection of some best practices (e.g., technical reviews) into the development process, or the adoption of a new development standards. On the other hand, the knowledge units may adopt the consultant type of relationship to provide support without disturbing the ongoing development. If the knowledge unit is dealing with technology domains (e.g., programming language, frameworks used in development), it is more suitable to adopt the consultant type of relationship, as the required level of support varies with respect to the particulars of the project, as well as the experiences of the project participants.

The realization strategy represents the third point of variation. A knowledge unit can be realized in the form of a separate department [24], or a position [116]. The last point of variation deals with the level of integration between the knowledge management units and project development units. The level of integration refers to the extent of change in the organization structure and culture required to accommodate the knowledge unit within the organization [98]. The overall organization structure, type of supported knowledge, and the objectives of the knowledge unit are the major factors affecting the selection of the suitable combination of these points of variation. For example, a knowledge unit supporting tacit knowledge may be realized as a position. In this case, the created position can be accommodated with minimal impact on the organization structure and culture. Transferring knowledge across geographical boundaries may require a mechanistic strategy. The impact of this specific organization structure and requirements of the knowledge unit may enforce the need for a separate department to capture and process the knowledge.

Despite their variations, all studies recommended the separation between the knowledge management unit and the development units; members of the knowledge unit should not be involved in the development organization. They should convey the knowledge to members of the development organizations and help them solve the problem rather than solving it themselves.

In this research we are interested in capturing knowledge in an explicit format as well as developing a systematic method to support the knowledge transfer process. We selected to define the knowledge unit as a centralized department, working as an internal consultant. We envisioned the centralized structure to be more suitable for our purpose for two reasons: (1) it supports the preservation of knowledge in explicit format and (2) it facilitates the incorporation of new knowledge domains in a systematic manner. Although we chose to define the knowledge unit in this thesis as a department, merging the identified tasks so that they are handled by one person can easily be achieved. Knowledge was provided through an internal consultant to minimize the cultural and social impacts of the knowledge unit. We have viewed the cultural and social aspects as falling outside the boundaries of the thesis. However, we used alert mechanism (such as lectures) to advertise the benefits of the knowledge packaged in our experience base.

2.4.1 Examples of knowledge units

By far, the Experience Factory [24] is the best documented approach practiced in industry (see for example [36], [10], [9], [145]). However, approaches like the Knowledge Brokers [44] and the Experience Engine [116] prove to be useful as well. In this section we will discuss these three approaches emphasizing their differences.

Knowledge brokers [44] are full-time employees in the knowledge unit. Knowledge brokers act as internal consultants, they support projects with their own knowledge and identify possible internal and external knowledge sources. Each knowledge broker is responsible for one topic (e.g., requirement engineering, specifications, testing). They are responsible for identifying best practices, and maintaining knowledge stores related to their specific topic. The concept of knowledge brokers, as described in [44], is an example of the decentralized structure, with a consultancy type of relationship. Knowledge brokers were realized as positions.

The experience engine concept [116] is based on maintaining "yellow pages" of available expertise. The engine defines two roles: experience brokers and communicators. Brokers are the visible members of the knowledge unit. Their role is to maintain the yellow pages, and facilitate the contact between experience communicators and those who need the knowledge. Experience communicators are those individuals who have the expertise. Their role is to help others solve their problems. The concept of experience engines, as deployed by Ericsson Software Technology AB [116], is an example of decentralized structure, with a consultancy type of relationship. In this deployment, the knowledge unit was realized as a department.

The experience factory is a logical and/or physical organization that supports project development by acting as a repository that captures the results of analyzing and synthesizing all kinds of experience and supplying that experience to various projects on demand [24]. The experience factory, as deployed in [23], was realized as a separate department. It is an example of centralized structure, with a more or less supervisory relationship with the development organization. Due to its importance, and more direct relevance to this research, we expand on the experience factory paradigm in the sections that follows.

2.5 The experience factory paradigm

To address knowledge management with the purpose of software quality improvement, Basili *et al.* [24] introduced the experience factory paradigm. The paradigm was initially introduced to support the reenactment of successful development activities. More recent work discusses tailored versions of the factory to support learning organizations whose main business is not software [19].

The paradigm organizes a software development organization into two organizations with separate goals: experience factory and project organization. The project organization focuses on developing and delivering software products and the experience factory focuses on improving development practices in the project organization by learning from experiences. The two organizations interact to support each other's objectives. The experience factory structure and its interaction with the project organization is conceptually represented in Figure 2.3 [24]. The factory is centered around an *experience base*, which contains an integrated set of packaged experiences that capture past development competencies. The factory supports three different organizational units that interact with the experience base: Support, Analysis and Packaging.

The *Support unit* facilitates the interaction between the factory and the developers. It saves and maintains the information in an easily and efficiently retrievable format. It also controls and monitors access to this information.

The *Analysis unit* processes the information received from the development organization to provide direct feedback to individual projects. It also produces, and may provide upon request, tools, lessons learned, baselines, etc.

The *Packaging unit* works off-line to generalize, tailor and formalize information and project experiences. It packages useful experiences in a variety of

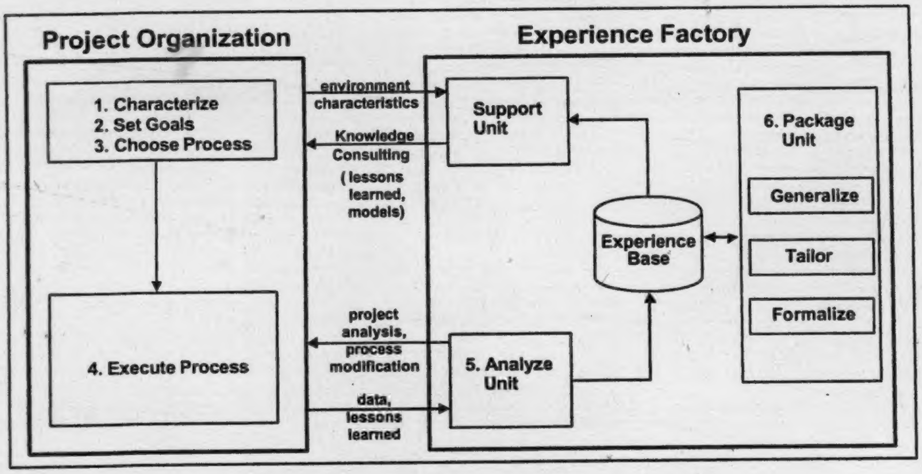

Project Organization
Experience Factory
1. Characterize
2. Set Goals
3. Choose Process
4. Execute Process
environment characteristics
Knowledge Consulting
(lessons learned, models)
project analysis, process modification
data, lessons learned
Support Unit
Experience Base
5. Analyze Unit
6. Package Unit
Generalize
Tailor
Formalize

Figure 2.3: Conceptual structure of the experience factory

models that meet the needs of different users of these experiences.

The experience factory institutionalizes the capture and utilization of the organization's knowledge to use it for collective learning purposes. To realize its purpose, it must support a set of interacting mechanisms for experience acquisition, packaging and evolution as well as methods for providing packaged experience to its potential users. Through these mechanisms, different experiences are collected, analyzed and then packaged in the experience base in order to provide, upon request, feedback to new projects based upon the experiences of similar projects. Packaged experiences may come from experimentation or previous development experience either local to the organization, or from the software industry at large. The experiences take into account the software discipline's experimental, evolutionary, and non-repetitive characteristics.

The operation of the experience factory is based on the Quality Improvement Paradigm (QIP) [21]. The paradigm is based on the notion that improving software processes and products requires continuous learning through accumulating experiences in well-understood forms and models that can be accessible to other projects to use and/or modify. QIP involves the following six consecutive steps:

1. **Characterize.** Understand the environment and understand the exist-

ing business process baselines.

2. **Set goals.** Based on the existing characterization and the capabilities of the organization, set quantifiable, reasonable goals based on current process baselines.

3. **Choose process.** On the basis of the environmental characteristics and the set goals, choose an appropriate improvement process and provide any required tools.

4. **Execute process.** Enact the process and provide feedback to measure progress against goals.

5. **Analyze process data.** At the end of the process, analyze all data collected, record findings, determine problems and make recommendations for future improvements.

6. **Package experience.** Consolidate the gained experience in a new (or updated) experience package from this and prior project experiences.

In a successful implementation of the experience factory paradigm, each project will follow its own process model. Choosing the right model will take advantage of process models provided by the experience factory to select the model that best fits the project's context and its product characteristics. Projects can access information about prior projects at different levels of abstraction, examining problems and solutions, effective methods, tools, etc. By accessing this prior experience, project managers can tailor the best possible processes, methods, and tools.

2.5.1 Discussion

The significant advantage of the experience factory paradigm lies in the transferring of development experiences, usually stored in the developers' minds, into permanent tangible corporate assets. The experience factory also provides necessary resources and expertise to support a wide variety of activities such as training, consulting, process management, process formalization, software measurements and evaluation as well as organization learning. However, the experience factory as implemented by Basili *et al.* [51] supports only codified knowledge; it does not provide a mechanism to handle tacit knowledge.

Although technology supporting the experience factory concept has been studied by many researchers [6] [102], successful realization of the experience factory concepts based on Basili's results is extremely challenging [222]. The challenges vary from defining exactly what constitutes an experience and, how it can be captured, documented and stored, to institutionalizing effective mechanisms to select the most relevant experience from the knowledge base [106]. For example, experience packages presented while explaining the experience factory concepts took the form of informal reports [23] [25]; yet, manuals of the first implementation of the experience factory indicate that the experience base was implemented using an online database [51]. Furthermore, the constructed experience base was criticized by its limited ability to incorporate knowledge from software engineering body of knowledge [174, Chapter 5].

Theoretical treatment of the experience factory created many of these challenges [106]; the treatment focused on explaining the factory and related strategies, giving little information about how experiences are accumulated, stored and used in day to day process management. Furthermore, Houdek *et al.* [107] indicated that these challenges existed even when the experience factory was documented fifteen years after it was first initiated. The description of how to establish the experience factory was written with hindsight, therefore it may be difficult to repeat.

2.6 Technologies to support the experience factory

Seaman *et al.* [202] identified three technical aspects that need to be considered for supporting the experience factory: repository, user interface and procedural. The repository aspect deals with how experience should be stored electronically. The user interface aspect deals with how to represent the experiences to its potential users and how the users will manipulate, search and retrieve experiences. General purpose browsers [97] and special interfaces such as the Visual Query Interface [209] have been used to support the factory's user interface [202]; however, many of the reports discussing the factory's implementation do not emphasize aspects of the user interface.

The procedural aspect focuses on knowledge management issues, such as how experiences are acquired, reused, maintained and updated. To date, most of the technologies developed to support knowledge management within the experience factory have focused either on knowledge acquisition [32] or knowledge deployment [103]; little support has gone into knowledge organization [196]. In the rest of this section we examine candidate technologies to support the procedural aspects of the experience factory. The discussion is organized around stages of the knowledge evolution cycle.

2.6.1 Knowledge acquisition

Technologies supporting knowledge acquisition have to overcome the very difficult task of making tacit knowledge explicit. The technology is usually built around a specific acquisition technique. Several techniques have been used to acquire software knowledge: literature surveys, expert consultation [75] and collecting all uses of the experience base [103].

Literature surveys help organizations to solicit knowledge from the software industry at large. This technique is useful to start a knowledge base, update the knowledge base with industry best practices or, to explore the usability of standards and recommended practices. When using consulting experts, a large set of solutions for a given problem should be solicited. The experts are then questioned about parameters that differentiate the solutions (solution space) [75]. This technique is useful for relatively mature knowledge domains.

The amount of knowledge relevant to the software process and the disagreement among experts about key parameters affecting process performance require the collection of information about concrete enactions of the process in order to build a knowledge base for an organization. By treating every enaction of the process as new knowledge to be acquired, a rich set of data can be collected, documented and analyzed. Several approaches such as informal reports [23], [25], structured text [106], [33] and formal languages [174] have been proposed in the literature to document and store enaction knowledge.

2.6.2 Knowledge deployment

Software experiences have been made available through a human consultant [105] and automated tools [97] [209] [202] [8]. For purposes of this discussion, knowledge deployment implies that the knowledge about a software process is codified, stored, and retrieved using an automated tool to support making decisions about the process.

Technologies to support knowledge deployment are based on the expected size of the Experience Base (EB). Lists and indexed catalogues have been used to document experiences in small-to-medium size experience bases [97]. For large experience bases, reasoning technologies are recommended [102] [7]; although, relational databases have also been used [202]. Three reasoning technologies can be considered in knowledge deployment systems: rule based, model based and case based.

Rule-based reasoning systems require the extraction of the domain knowledge and encoding the knowledge into rules. Each rule contains a small chunk of information and reasoning is done by rule composition. The intuition is "by iteratively applying the knowledge rules, answers to questions may be derived." With little guidance about rule contents, expressing knowledge into rules is not a trivial task [94]. Evolving a rule based system is not a simple task as well [212], because changing one rule often requires modification of several other rules. Rule-based reasoning implies that most of the domain knowledge is known and can be encoded into rules; an implication that does not generally hold true for the software process domain.

Model-based reasoning derives answers by knowing the causal model of the domain. Models tend to hold information needed for validation or evaluation of the solution, but do not provide methods of constructing the solution. For example, models for human resource management in software projects [2] provide some insight into Brooks' Law[1] [43], but they stop short of suggesting balanced manpower acquisition policies to overcome this problem. The underlying paradigm in the competency refinery supports mechanisms for constructing solutions. Hence, model-based reasoning alone is not enough for supporting the refinery concept. Furthermore, reasoning using causal models assumes that the domain is well enough understood to enumerate a causal model. Very few software organizations understand their software processes well enough to reason about them, a state that limits the applicability of model-based reasoning at present.

---

[1] Brooks' Law states that "Adding manpower to a late software project makes it later."

Case-based reasoning systems draws decisions on the comparison between remembered cases and the new situation. The intuition is "what has been done before to successfully solve a problem may be successfully used in similar situations." They typically reason using large chunks of knowledge, rules and similarity metrics for adaptation - a type of knowledge that is easier to acquire. Due to the importance of Case-Based Reasoning systems to this thesis, an overview of how it can be deployed is given in Appendix A.

2.6.3 Knowledge creation and organization

Knowledge creation and organization add context to the information captured by the system or imported from external sources. Knowledge creation and organization tasks include maintaining the knowledge base according to a specific classification, adding new relationships between knowledge items, setting up a hierarchy of knowledge items and maintaining historical data about the usage of the knowledge items. The level of support that can be provided at this stage depends on the nature of information processed. For example, little support can be provided for knowledge creation by soliciting information from the software industry at large, because such information is presented in an informal manner, often as a report or a working paper. Extracting knowledge from these reports is typically a human intensive activity. On the other hand, qualitative research methods and statistical analysis techniques can be deployed to support knowledge organization for formal and semi-formal data captured during the usage of different knowledge items. For example, the effect of a new testing tool can be assessed by interviewing testers in the organization or by comparing the defect detection rates before and after the tool.

2.7 Implementation of the experience factory concept

Most of the reported implementations of the experience factory focused on the software process. However, efforts in building domain specific experience factories in the areas of data mining applications [17], developing CBR applications [11], and ontology deployment [125] have also been reported.

The first implementation of the experience factory was at NASA Goddard Space Flight Center (GSFC). The center established the Software Engineering Laboratory (SEL) in conjunction with University of Maryland and Computer Science Corporation. SEL was established in 1976 to support research in the measurements and evaluation of software development process. One of its major responsibilities was the collection, storage and archival of software engineering data. In this environment, the experience factory concept was proposed, developed [24] and enacted [23]. SEL maintains its data in an online database [51] implemented using the ORACLE Database Management System. Over a ten year period, the experience factory at SEL managed to reduce development cost by 60%, decrease error rate by 85%, and reduce cycle time by 20% [202].

The success of the experience factory at SEL has motivated other organizations to build their own experience factory. Houdek *et al.* [107] reported Daimler Chrysler's initial experiences in establishing experience factories. Three different projects formed the basis of the analysis: the first two initiatives focused on formal reviews and the third dealt with acceptance processes. The study concluded by defining a plausible mandate for similar initiatives [200], and recommending more studies to validate and generalize their observations.

An Australian telecommunications company established an experience factory [135] with the goal of enhancing the transfer of process knowledge amongst projects. The factory was built by providing an effective framework for access and integration of the information already existing in the organization's repositories. Despite achieving its technical objectives, the system was decommissioned shortly after the completion of the project. The lack of ongoing management commitment and the lack of identification of clear goals and payback criteria was among the reasons contributing to the decommission.

Chatters [52] reported on ICL's efforts to develop a framework to support its method engineering based on the experience factory paradigm. The framework has four key components: tools development, learning, deployment and experience. Checklists and process descriptions are examples of the tools developed in the tools component. Through the learning component, project members are trained on how to deploy the predefined tools. Finally, the experience component captures the results, an assessment of effectiveness of specific applications of the tools, and any lessons learned. Information gathered from the application of the framework is captured in a knowledge sharing repository.

Recently, Fraunhofer Institute for Experimental Software Engineering built a COrporate Information Network (COIN) to facilitate experience sharing among different projects within the organization. COIN was used to develop and validate a goal-oriented experience management approach. It consists of three main parts: the experience base, the COIN Team and an intranet representation [8]. The project is focused on business process description and lessons learned.

Until recently, there is a limited number of published enactions of experience factories. These publications focused either on the structure of the factory as in [107] and [52] or the structure of the experience base as in [8]. Methods to create, organize and evolve knowledge within the experience factory are rarely discussed; details of the experience base to support this evolutionary nature of knowledge are still vague. Furthermore, publications dealing with the experience factory did not provide a comprehensive description for the organizational units inside the factory nor how these units work together and interact with the development organization.

2.8 Summary

This chapter presented an overview of the measures taken by Learning Software Organizations to capture and disseminate software engineering knowledge. Typically, LSOs have to deal with two fundamental issues: knowledge management and organizational infrastructure. We identified three main categories of software engineering knowledge to capture for a software organization: domain, methodology and technical expertise. Strategies to manage this knowledge depends on its format: explicit or tacit, as well as the objectives of the organization. We also explored different organizational structures used to support LSOs. Finally, we turned our attention to the experience factory paradigm discussing its merits, problems of reenactment, as well as technologies developed to support it.

Having investigated the fundamental issues facing LSOs, a number of weaknesses were identified. It was decided to build an experience factory focusing on software process expertise. The objective is to focus on the process of starting and running one rather than its structure. We primarily want to:

- define process patterns for structuring and storing process experiences. The pattern should be capable of representing hands-on experiences and support the evolution of these experiences to recommended practices and lessons learned.
- provide mechanisms to support a combination of mechanistic and organic strategies for knowledge management within the factory. However, the ultimate goal is to capture explicitly the required knowledge and support it mechanistically.
- document the processes and infrastructure required to create, evolve and disseminate knowledge within the organization.

These issues were identified as being vital to the success of starting a Learning Software Organization. In order examine the validity of our ideas and concepts we built an experience base for technical reviews and ran the factory to support a senior class in software engineering at the Department of Computing Science, University of Alberta. In the next chapter we discuss our proposed structure of the factory as well as the templates we use to capture experiences.

Chapter 3

A Model for Selecting Process Steps

3.1 Introduction

In this chapter, our model for knowledge management in software, called the Competency Refinery (CR), is presented. The model is based on the experience factory paradigm proposed by Basili *et al.* [24]. It supports the capitalization and reuse of development experiences. The main goal of the competency refinery is to facilitate quality improvements by analyzing and packaging software development competencies.

The main function of the competency refinery is to collect, package and maintain process experiences and evolve and continuously improve the experience models. Because the competency refinery is built on the concepts of the experience factory, it utilizes the same quality improvement paradigms and adopts the same principle of separating the factory organization from the project organization. It further improves the paradigm by centralizing and focusing experiences on knowledge units [52].

The refinery supports two types of users: technical leaders (or project managers) and quality improvement engineers. Technical leaders use the refinery to make more knowledgeable decisions about the development, whether they are facing circumstances that are new or similar to previous projects. Quality improvement engineers use the refinery mechanisms to extract tacit knowledge and present it in an explicit format. The extracted knowledge can further be used to improve the performance of the technical staff.

In the rest of the chapter we further discuss the competency refinery model. The next section presents the basic concepts and paradigms used in the refinery, explaining how it differs from the experience factory. Sections 3.3 and 3.4 discuss the refinery's architecture and the details of the knowledge base as it applies to the software process domain respectively.

3.2 The Competency Refinery concepts

The Competency Refinery paradigm supports the reenactment of successful development activities and institutionalizes continuous improvement through the capture and utilization of the organization's knowledge and collective learning. It provides a mechanism for improvement through creating, analyzing and packaging software development competencies within the organization. Competencies are created by documenting development experiences within the organization, by importing generally accepted software engineering practices, or through the innovation in development activities within the organization. After their creation, competencies are continuously refined by deploying them in development. Experimentation serves as an important technique to create and refine competencies as well.

The paradigm can be thought of as an extension to the experience factory paradigm presented by Basili *et al.* [24]. The major differences are in the overall objectives and the nature of stored knowledge. The paradigm has evolved from a software environment concerned with storing software artifacts for reuse, to an environment to store development experiences for the purpose of detecting development competencies existing within the organization and exploiting their intrinsic benefits.

While the main objective of the experience factory is facilitating software reuse through packaging knowledge supporting software reuse, the major objective of the Competency Refinery is to package knowledge to support decision making and facilitate organization learning. The Competency Refinery capitalizes on the evolutionary nature of knowledge. Experiences from deploying

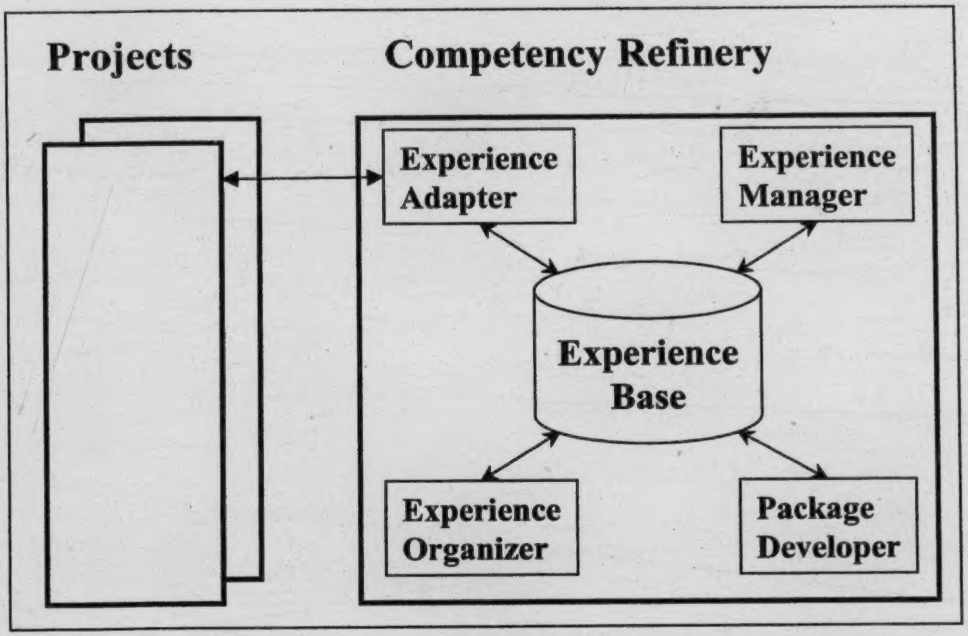


Figure 3.1: The Competency Refinery

stored competencies in other projects lead to capturing more of its tacit part. Through this process, issues affecting the quality of the competency may be identified for the purpose of quality improvement and/or better control.

Experience bases built in most implementations of the experience factory store one class of knowledge; some implementations focus on processed information [22] [222], others focus on hands-on experiences [102]. Objectives of the competency refinery mandates maintaining historical data at different levels of abstraction. To accommodate these requirements, different types of experience packages are maintained to capture different classes of knowledge about the competencies. In the competency refinery, the main source of knowledge is hands-on experiences. By collecting and storing hands-on development experience, the Competency Refinery creates and maintains a knowledge base about the competencies administered by the refinery. By continuously analyzing and synthesizing these experiences, lessons learned from its usage are abstracted and stored. Models describing the competency and its performance may also be developed by consolidating different hands-on experiences.

The Competency Refinery is conceptually represented in Figure 3.1. The refinery is centered around an *experience base*, which contains an integrated set of packaged experiences that capture past development competencies. To serve its purpose the competency refinery supports a set of interacting mechanisms for experience acquisition, packaging and evolution as well as methods for providing packaged experience to its potential users.

3.3 Competency Refinery architecture

An implementation of the Competency Refinery is composed of contents, architecture and tools. The contents sustains the core value of the refinery, namely the experience packages. Packaged experiences can be raw data collected from various projects or knowledge extracted by abstracting the data. Through out the thesis we use experience and package interchangeably to mean experience package. Tools support managing and communicating the contents of the refinery. The structure is the foundation that enables the refinery to serve its purpose. The refinery structure proposed in this thesis is composed of two parts: reference architecture and interaction mechanisms. The reference architecture, as defined in [31], describes agents of the structure and their necessary communication paths, leaving the particulars of their implementation to the refinery instantiation. Therefore, there are no assumptions about the way these components may be implemented. For example, they can be implemented using human or computer based systems.

Interaction mechanisms define activities required for the components to function on daily basis. Through the set of interaction mechanisms supported by the competency refinery, experiences are collected, analyzed and then packaged in the experience base in order to provide, upon request, feedback to new projects based upon the experiences of similar context. These mechanisms can be grouped in three different sets of activities to interact with the experience base: *identification, storage* and *communication* [226].

Communication activities facilitate the interaction between the refinery and its users, with the objective of controlling and monitoring access to in-

Agent	Specification	
Experience Manager	Activities	Package storage Package retrieval Taxonomy management
	Comm. Paths	Experience adapter Experience organizer Experience developer
Experience Adapter	Activities	Package selection Project consultation Feedback generation
	Comm. Paths	Experience manager Experience developer
Experience Organizer	Activities	Information consolidation Knowledge formalization
	Comm. Paths	Experience manager
Experience Developer	Activities	Experimentation Package generalization Package composition
	Comm. Paths	Experience manager Experience adapter

Table 3.1: Activities and communication pathes for the competency refinery

formation stored in the experience base. Experiences captured from different projects, or other relevant sources of experience, are processed to identify new experiences that need to be packaged. Through identification activities, captured experiences are abstracted to establish baselines (e.g., defect rates), build models (e.g., quality) and designate definitions (e.g., process). Storage activities focus on the consistency and diversity of the captured experienced.

3.3.1 Reference architecture for the Competency Refinery

From the discussion of the sets of activities associated with the refinery we define requirements for four architectural agents. Activities performed by these agents and their possible communication paths are summarized in Table 3.1. The agents are:

1. ***Experience Adapter.*** The experience adapter is the main interface between the refinery and its users. It selects and tailors a coherent set of experiences that satisfy the project requirements. Selected packages are based on the knowledge accumulated in the experience base. The adapter is also responsible for tracking the deployment of selected experience packages and documenting the resulting hands-on experiences. Generally, the experience adapter is the only agent actively interacting with the production process, by feeding knowledge and collecting experiences.

2. ***Experience Manager.*** The experience manager controls the knowledge that resides in the experience base. In addition to storing the experience packages, the manager responsibilities are addressed by two activities: structural management and content management. Structural management extends the taxonomy of the experience base to include new areas of knowledge according the organization needs. Content management is concerned with the integrity of knowledge within the experience base. The experience manager is concerned with the syntactical aspects of the experience base. Access control and access strategies are also main functions of the experience manager.

3. ***Experience Developer.*** The responsibility of the experience developer is to create new experience packages. New experiences can be developed by adapting, generalizing and/or assembling pre-existing experiences, or adopting public domain processes typically found in the software literature. The developer agent is also responsible for validating created experience through practical techniques (e.g., experimentation). The experience developer also defines measurements that need to be collected to assess the value of the newly deployed experiences or to support knowledge area expansions proposed by the experience manager.

4. ***Experience Organizer.*** The responsibility of the experience organizer is to maintain experience packages already existing in the experience

base. The organizer agent develops models to analyze existing experiences or validate the applicability of existing models by examining them using newly acquired data. The semantics of the packaged experiences is the main concern of the experience organizer.

Activities of the experience organizer are asynchronous with respect to the production process in the organization as it interacts with the project organization through the experience base. According to the QIP, knowledge packaged by the organizer agent start with a simple model that is incrementally enhanced in order to improve and/or expand the capabilities of the packaged competencies.

This list of agents represent a complete set of architectural agents that cover all the activities related to the knowledge evolution cycle stages (see Figure 2.1), as explained in Figure 3.2. Agents of the refinery interact together either to communicate experiences or to extend the experience base. Communication paths identified in Table 3.1 support experience-base extension.

3.3.2 Instantiation of the architecture

In order to instantiate this architecture, interfaces of the architectural agents have to be defined as well as the flows of data and control among the agents. This implies finalizing the agents' communication specifics and the distribution of control among agents. Similar to the experience factory, the refinery can be started following two possible approaches: top-down and bottom-up [25]. That is, proceed either from a well-defined ontology, to a schema for the experience base, then collect concrete experience data, or else collect concrete experiences and proceed towards abstracted knowledge. However, the bottom-up approach mandates the definition of a taxonomy to start data collection. In this implementation of the refinery, we favored the bottom-up approach over the top-down for these reasons:

- Top-down approach assumes a relatively stable environment [5]. This assumption does not hold true in general, as many organizations favor

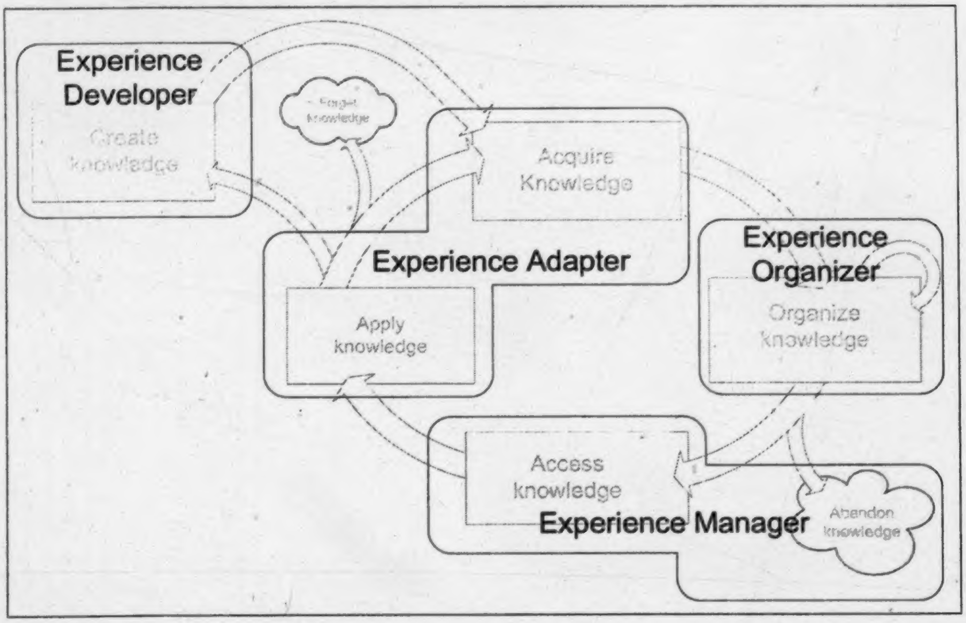

Experience Developer
Create knowledge
Forget knowledge
Acquire Knowledge
Experience Adapter
Apply knowledge
Experience Organizer
Organize knowledge
Access knowledge
Experience Manager
Abandon knowledge

Figure 3.2: Relation between the competency refinery and the knowledge evolution cycle

and, indeed live with, dynamic development environments to cope with short technology cycles.

- Our approach focuses on data collected from hands-on experiences. Most of documented experience factory implementations following the top-down approach are either based on long-term application [23], or experimental data [11].
- For simplicity, we started with the simplest solution that would work as recommended by newer software development methodologies [27].

From his experience in building knowledge units, Schneider concluded that [200] seeding the experience base is a fundamental requirement to start a knowledge unit.

To limit the thesis scope, we chose to focus on the software process knowledge. Following the concept of knowledge units requires partitioning the experience base into knowledge domains (or topics). Knowledge captured in each knowledge domain is maintained independently. The knowledge unit we implemented in this thesis is focused on peer reviews.

In our enactment of the refinery all agents of the structure were implemented using human agents. However, we developed several automated tools to support the functions of the Experience Manager, the Experience Adapter and the Experience Organizer. Details of this implementation will be further discussed in chapters 4, 5 and 6.

3.4 Packaging software process knowledge

For an organization to benefit from a competency refinery, details about the experience base need to be well understood. Key questions that need to be answered include: What constitutes a process experience? How can it be captured, documented and stored? What motivates the selection of a particular process alternative?

In general, any knowledge related to the software development process is considered a process experience that could be captured, documented and stored for further use. To achieve the required learning goals, process knowledge needs to be kept in a normative or prescriptive format; a process package should prescribe how a sensible agent should act to achieve a certain goal. A process package may also be descriptive, describing the enaction details of a particular process. The following are a few examples of these experiences:

Hands-on experiences: Objects describing knowledge gained through process enaction. This knowledge can be described in natural, or semi-formal languages.

Process definitions: Definition of the process capturing its important details. These details include its inputs, expected outputs, participants and their roles, entry/exit criteria, etc. Process definition may be written in a language that may be formal, semi-formal, or graphical.

Development models: Process models obtained by aggregation of several processes to serve a particular purpose. These processes are integrated to cover one or more aspects of the development process. This includes quality assurance processes, testing processes, and life-cycle models (e.g., waterfall, spiral).

Assessment models: Objects describing how to measure the process, judging its success or failure. This includes mathematical models, process standards.

Supporting documents: Textual objects written in natural or semi-formal language with figures, tables, checklists to communicate information in some organized way, (e.g., hypertext objects). This includes specifications of documents produced or consumed by a process, data analysis techniques, recommendations, reports from specific studies and analysis, etc.

Process experiences produced and maintained in the process refinery are called Process Packages. Each process package is a composite object made of one or more process experiences. The capability of the process refinery to organize and synthesize process experiences is a critical element for the successful support of process improvement.

3.4.1 Types of process packages

In the context of this thesis, process experience is viewed as the *"practical knowledge or skill abstracted or directly observed from participation in a particular activity"* [161]. We are interested in capturing the <abstracted knowledge, direct participation> tuple. This tuple implies that we are focused on packaging knowledge rooted in participation and accumulated during everyday work in the organization. However, knowledge and participation need not be reported in the same package. In fact, we view participation reports as the concrete knowledge from which abstract processes may be deducted.

The <knowledge, participation> tuple implies that development knowledge can be divided into *concrete* and *abstract* aspects. Concrete knowledge captures hands-on experiences and abstract knowledge is a generalization of the concrete knowledge. It supports the decision making process of a project by offering packaged solutions to its problems. This knowledge may come from in-house experiences, experimental results or the adoption of standard practices in the software industry. In despite of the basis of the abstract knowledge, a critical aspect of process improvement within the organization is the continuous refinement of the abstract knowledge using related concrete experiences.

Process experiences need to be packaged in a variety of ways to fulfil different interests of its users. For example, during project planning, experience base users are more interested in exploring options to decide on the set of processes to use. At this stage, they are interested in process merit and major risks, inter-process interactions and trade-offs, rather than how to enact it. When a particular process is chosen, a user's interest shift to issues like comparing the different methodologies to enact the process, and how to measure its success or manage its risks.

While concrete knowledge can be packaged as one type (*concrete* type), abstract knowledge needs to be packaged differently to fulfill different users' interests. To emphasize these differences we have chosen to package abstract knowledge as either: *praxis* and *modus* types. The three experience package types can be described as follows:

Praxis. Praxis packages document industry best practices, as well as, the best practices accepted within the organization. Praxis packages are general in nature, documenting for example, the efficacy of a process, the merits of a tool, with enaction details removed (i.e. abstracted out).

Modus. Modus packages focus on the details of a particular process or best practice. A modus package may document a particular methodology for enacting the process and, as necessary, clarify how to perform its sub-processes.

Concrete. Concrete packages are tightly related to the real world; they document hands-on experiences. A concrete package reports on how an abstract package is enacted in a given organizational context, and whether

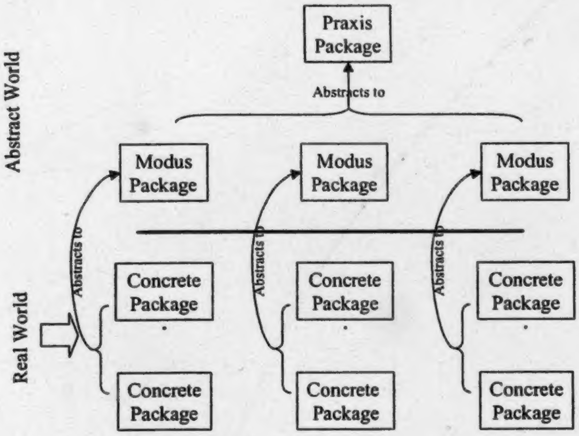

Praxis Package
Abstracts to
Abstract World
Modus Package
Modus Package
Modus Package
Abstracts to
Abstracts to
Abstracts to
Real World
Concrete Package
Concrete Package
Concrete Package
Concrete Package
Concrete Package
Concrete Package

Figure 3.3: Different levels of experience packages

the practice is a success or a failure. The package may include a recommendation of "what to do and/or avoid".

Generally, praxis packages capture the merits of industry's best practices; modus packages represent methodologies of enacting these practices and concrete packages describe the experience gained by participation on process enactions.

The three package types align with proposed models of process evolution [58]. The three package types can be viewed as representing development experiences at different levels of abstraction as depicted in Figure 3.3. Each enaction of the process is acquired as a concrete package. By analyzing a set of similar concrete packages, environment particulars are abstracted out and the knowledge is represented as one modus package. Details of the enaction methodology are further abstracted out to be documented as one or more praxis packages representing a best practice within the organization. For example, various methodologies of performing technical reviews (e.g., Fagen inspection [77] and IEEE standard review [112]) are represented as different modus packages. However, the merits and risks of technical reviews (despite the particulars of the methodology) are represented as one praxis package.

3.4.2 Experience Representation

The complexity of the software process domain presents several problems in determining and representing experience packages. At the macro level we must address the question: "What is the proper level of granularity for an experience package?" At the micro level, the main question is: "How to characterize different experiences?"

An experience package may represent a process-step (e.g., inspection kick-off meeting), a process (e.g., technical review) or a complete development methodology (e.g., eXtreme Programming [154]). At the concrete level, process methodologies are too general to prescribe in one package. On the other hand, we anticipate that process-steps are too specific. Hence, we set the granularity level of our experience packages at the process level. A process is defined as [126]: *a set of process-steps with well defined roles and input/output work products to serve a set of common objective.*

The core of a process package is the <*objective, prescription*> tuple. Process objectives are the key characteristic that set processes apart; in a sense they represent the "problem(s)" addressed by the process. The prescription part of the package details the "know-how" or the core knowledge about the process. Each process package must have at least one objective or goal to achieve. For practical considerations, no other constraints are imposed on process packages; for example, it is acceptable to include a process package without well defined input/output or missing a clear definition of roles, etc.

It is evident that the performance of successful process experiences is not globally consistent [208]; however our goal is to uncover experiences that provide consistent performance with respect to their enaction environment. Hence, reporting process prescription alone is not enough, the *context* of the knowledge contained in a process package is also important to report, specially for concrete packages.

The goal of the experience base is to categorize packaged experiences based on certain *features*. A typical feature may be a chance of success at first enaction or a type of training required to perform the process successfully.

Process features may not be clear at first; they are determined following an iterative enactment of a process. Selection of new features to represent a package is determined by studying the set of available experiences, studying the discriminatory power of the selected features, modifying them if necessary, then starting the next iteration. For example, by analyzing concrete packages for inspection we might find out that formal inspection training increases the chances of success of the inspection process. Hence, "requires formal training" would be added as a feature for the inspection modus package.

Other information about the process (e.g., references, comments) to provide further information about the process is also required in the package. During software production, processes interact in a variety of ways. Details of this interaction and dependencies is captured in the process package as *related experiences.* For management purposes, each package should have a name, type, etc. The information in a process package is reported in a structured text format following an *experience package template* as illustrated in Figure 3.4. As the experience base matures, the discriminatory power of process features matures through a clear understanding of the network of interactions and dependencies among processes.

3.4.3 Selection criteria

After matching the context of the process alternatives, project managers have to apply some discriminate measures to select an alternative. The selection measures are usually based on the the cost-benefit relations of the available alternatives. Hence, a process package should contain cost-benefit models for the process. Cost-benefit models define the set of metrics required to assess the benefits of enacting the process as well as its set of cost drivers. Direct process costs can be determined quantitatively. Measuring or estimating the direct benefits is not so straight forward, as not all process benefits can be quantified. Hence, in order to evaluate process alternatives, qualitative criteria may be used.

Name: a unique identifier for the experience.

Type: Praxis, Modus or Concrete.

Objective: a list of the objectives satisfied by the process documented in the package.

Prescription: is a detailed description of the experience knowledge.

Context: characterization of the environment from which the experience was acquired.

Features: features of the experience that make it distinctive from other experiences in the experience base.

Related Experiences: listing of experience packages semantically linked to current experience (e.g. uses, contains), as well as information for navigation among experiences (e.g. linking inspection Modus package with corresponding inspection concrete packages)

References: Additional material discussing the experience (books, articles, manuals, etc.).

Comments: any additional information important for using the experience.

Administration: listing of administrative information.

Figure 3.4: Experience package template

3.4.4 Experience Acquisition

Closely related to experience representation is experience acquisition. Where to acquire experiences? When to say that available experience packages are enough for the organization's needs? There are three basic ways to gain experiences [33]:

- Use available technical knowledge sources;
- Use goal-oriented knowledge acquisition; and
- Accumulate knowledge during everyday work.

Concrete packages are typically internal to the organization. They document the enaction of a process (either in a project or in an experimental setting). Aspects of the process that can't be explored through day to day work are usually investigated in experiments designed to achieve this goal. Depending on the type of experiment, praxis and modus packages may be created as well. For example, praxis packages can represent the result of simulation experiments. Praxis and modus packages may come from internal or

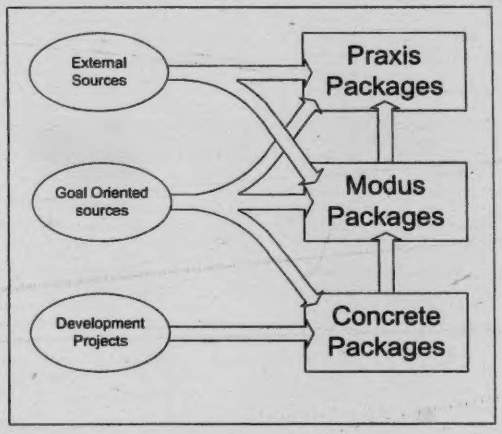


Figure 3.5: Sources of knowledge for different types of packages

external sources. If praxis and modus packages are internal, they are generated by abstracting enaction details from different concrete packages. However, relegating the praxis and modus packages to this path is overly restrictive, as it neglects the available knowledge accumulated in the software industry. External packages are acquired primarily from software engineering publications and standards. Figure 3.5 presents sources of information for all types of the process packages. Throughout the rest of the thesis the identified sources of abstract experiences are explored. In chapter 4 we explore aspects of generating knowledge packages from external sources. Documenting development experiences is discussed in chapter 5 and packages generation from experimentation is discussed in chapter 6.

3.5 Summary

This chapter presented our proposed architecture for the competency refinery. While the architecture is based on the experience factory concepts, the major differences are in the overall objectives and the nature of stored knowledge. The main objective of the experience factory is to store software knowledge to support reuse, the main objective of the competency refinery is to package knowledge to support decision making and facilitate organization learning. To serve its objective, the refinery is equally concerned with knowledge acquisition, evolution and delivery.

The architecture defines four agents centered around an experience base: experience adapter, experience manager, experience developer and experience organizer. Experiences are packaged in the experience base in the form of structured text that is tolerant to incomplete information and the incorporation of knowledge from software engineering at large. The next three chapters document our efforts and experiences in implementing and running a refinery for peer reviews. The first step towards the implementation, discussed in the next chapter, is to build an experience base for peer reviews.

Chapter 4

An Experience Base for Peer Reviews

4.1 Introduction

The main doctrine of the competency refinery is to support the selection of the process alternative that best suites the specific needs of the project under development. It provides the mechanisms to build experience bases and make available relevant experiences to different projects based on some similarity analysis.

Schneider [200] argues that a successful experience factory can not start with an empty experience base. The experience base needs to provide useful seeds [81] from the very first hour of usage. Process seeds could be driven from internal sources (e.g., the organization process manual) or external sources (software process literature). Unfortunately, most of the publications discussing experience factories, discuss either the specifications of the experience base (e.g., [8]) or how the experience base is used (e.g., [107]), ignoring the transitional step where the experience base is actually built and seeded. In this chapter we will discuss how process seeds can be constructed systematically by building an experience base for peer reviews [84].

Unfortunately, inconsistencies among publications addressing peer reviews, makes it difficult to identify similarities and differences among proposed and/or practiced review processes [140]. Hence, it is not easy to evaluate and reconcile the results for software practitioners. In order to identify those processes that can be used as seeds in the experience base, we need to integrate published work about reviews into a coherent body of knowledge through a comprehensive taxonomy. Because review literature can include many variations and ambiguities, this integration is also needed to assist software organizations in evaluating and benefiting from any research effort in the area [193].

First, we will review some of the proposed review taxonomies in Section 4.2. Some background about peer reviews is presented by describing some of the major research results in the area in Section 4.3. Section 4.4 discusses the process taxonomy we used along with its objectives. Peer review literature is discussed in the context of this taxonomy in Section 4.5. The chapter concludes with a summary of our key findings.

4.2 Related work on taxonomies for peer reviews

The importance of peer reviews coupled with the ambiguities among published work stimulated many trials to consolidate and reconcile peer-reviews research findings. At a high level, peer reviews have been classified based on process formality and objectives into inspections, technical reviews and walkthroughs [108] [84]. Inspections are the most formal process with the most precise objectives. Walkthroughs are the least formal with the widest range of objectives. According to this classification, walkthroughs are used for training, technical reviews are used for consensus formation and planning, and inspections are used for improving the quality of software artifacts.

At a lower level of detail, surveys provide taxonomies to classify peer reviews based on different subsets of its attributes. These attributes include [180] [237] process objectives, input/output characteristics, required preparation, collection techniques, team size and member roles, number of sessions, how the sessions are coordinated and defect detection method. Unfortunately, most of these studies looked only at formal — inspection like — reviews excluding and/or marginalizing some useful review methodologies (e.g., cognitive walkthrough [236], IEEE standard technical review [112]). In a general sense, contributions of available surveys can be summarized as follows:

- Kim *et al.* [131] classified reviews across five dimensions: aims and benefits, human element, process, output and other matters. The survey focused on how to perform a peer review.
- Macdonald *et al.* [148] focused their classification on the process. Their goal is to formalize the process for the application of tool support.
- Porter *et al.* [180] focused on the review process that are geared towards defect detection. The survey identified variation points among different review methodologies paying attention to the costs and benefits of methodology alternatives.
- Wheeler *et al.* [238] divided reviews based on number of review participants. They further categorized reviews with limited number of participants into inspections, walkthroughs, selected aspect reviews and others. The survey focused on elaborating the differences between inspection and other peer review processes.
- Tjahjono [225] mapped formal technical reviews (FTR) methods into a series of phases. Each phase is described in terms of seven basic components: objective, collaboration, roles, synchronicity, technique, entry and exit criteria. The survey's goal is to determine the similarities and differences between different FTR methods.
- Laitenberger [140] classified software inspection along four dimensions: technical, economic, organizational and tool support. The goal of the classification is to articulate the core concepts and relationships of software inspections. Although Laitenberger's model tried to be global, it failed to classify reviews in orthogonal dimensions. Confusion was evident when attributes of different dimensions were defined.

These surveys helped in identifying potential success factors controlling a peer review process. Unfortunately, most of available surveys evaluated only a subset of these success factors. Hence, resulting taxonomies are limited to the perspectives and objectives of the study — for example, [180] is part of a study [213] looking at cost-benefit analysis of inspections. On top of that, most of these surveys either ignored the organization context or project specifics while presenting their findings. This makes it difficult for software developers to choose the proper review that would fit their needs, and thereby provide maximum impact on specific projects for the development organization.

4.3 Background on peer reviews

Peer reviews have been practiced in the software industry for over twenty-five years. They are used primarily for the detection and elimination of defects in software artifacts as soon as these artifacts are created [77]. The core process of any proposed and/or practiced peer review involves a team of independent experts examining software artifacts. The benefit of reviews is supported by the argument that it is easier to detect errors in someone else's work than in your own; a phenomenon known as 'cognitive dissonance' [232].

In general, a peer review process has three stages: preparation, examination and follow-up. Specifics of these stages vary greatly depending on the process objectives. With the objective of defect detection, Fagan inspection [77] represents one of the earliest formal peer review processes described in literature. Fagan inspection stimulated a substantial body of work in peer reviews over the past twenty years. The published work concerned with peer reviews can be categorized [1] into: *empirical studies* and *experience reports.*

4.3.1 Empirical Studies

Although there is a general agreement about the key factors affecting the success or failure of a peer review, the contribution of each factor towards review success is not well understood. For example, Eick *et al.* [73], Weller [234] and Tjahjono [225] reported that the larger the number of reviewers, the better the review performance; in Eick's report [73], the performance increased

---

[1] These categories are not mutually exclusive, a paper may present an enhancement in methodology supported by industrial experience data (e.g., Fagan's Inspection paper [77]).

Method	Efficiency	Document type	Team size	Ref.
Paragraph effect analysis	37.3%	code	1	[104]
Code walkthrough	38%	code	3	[168]
Phased inspection	50%	code	2	[134]
Fagan inspection	20% - 46%	code	4 and 7	[69]
FTArm	46.4%	code	3	[225]
N-fold inspection	27%	requirements	4	[155]
N-fold inspection	35.1% - 77.8%	requirements	4	[199]
Inspection	25% - 50%	requirements	3	[184]

Table 4.1: Empirical results of peer review experiments

by 600% for 8 persons review team over individual reviewers. On the other hand, Bisant and Lyle [34] found no difference in performance between 3, 4 and 5 person review teams.

To understand the causal factors underlying peer reviews success and effectiveness better, many researchers conducted carefully controlled laboratory experiments. Most of these studies focus on the defect detection aspect of peer reviews. Study goals varied from evaluating the process as a whole (e.g., peer reviews against testing [104] [168]) to evaluating details of the process (e.g., comparing the effectiveness of a set of defect detection techniques [183] [225, 184]). Goals sometimes included an investigation of the superiority of a specific review process [199] [134].

Reported experimental results, summarized in Table 4.1, shows a wide variations in review performance. Reported defect detection efficiency[2] ranges from 20% to 50% for code and 25% to 77.8% for requirement documents. There is no definitive explanation for this wide variation, however, factors include review method used, team size and/or the document type.

4.3.2 Experience Reports

While introducing his inspection process, Fagan [77] demonstrated the process efficiency by comparing results pulled from the implementation of a large operating system project. Fagan chose two pieces of a moderately complex component. One piece was subjected to inspection after the detailed design and the coding phases, the other piece was not inspected at all. Fagan reported an increase of 23% in coding productivity, a saving of one programmer month per KNCSS (1000 Non Commented Source Statements). He also reported quality improvement as the inspected piece contained 38% less errors.

---

[2] In this context, defect detection efficiency is defined as percentage of total defects found.

The promised cost savings and quality improvements in Fagan's report promoted the enaction of peer reviews in many industrial setups. Reported industrial experiences focused mainly on discussing peer reviews benefits and limitation. Review benefits are reported as either improvements in product quality or reduction in development time. Limitations on the other hand, are presented as problems hindering the deployment of a successful peer review process.

Industrial reports show that code is the most reviewed work product in industry [84] [46] [129]. Review of design [100] [160] and requirements [67] artifacts is also reported. Furthermore, industrial experiences are usually discussed in the context of a particular review process experienced by an organization. However, details of reported review process have to be examined carefully. Many organizations which claim to use Fagan inspection are not using the process as specified by Fagan [92].

Despite the number of reported successes, deployment of peer reviews is not always successful [208]; Brykcysnki [45] attributed this to industry's frequent failure to adopt a successful peer review process. In the literature, this failure was attributed partly to problems with enacting the technical details of the process and partly to the development environment.

The major reason of failure related to reviews deployment is enaction errors [110]. In one survey [92], 84% of surveyed organizations claimed to perform Fagan inspection, yet none of them performed it exactly as specified by Fagan. The relatively high process cost, setup and running costs, provides another reason for failure, specially if review data is not well managed [110] [93] [45].

The biggest problem with review enactment as observed by Humphrey [110] [111], is management inattention and schedule pressure. Other identified organizational problems include technology transitions [45] and the difficulty to

Organization	Quality Imp.	Saving	Method
Aetna	83%		[77]
Sperry Univac	27:1		[100]
IBM, Santa Teresa		1:20	[190]
IBM, UK	93%		[78]
Standard Bank	> 50%		[78]
AMEX	> 50%		[78]
IBM	85%		[169]
Banking Services firm		1:[2.2-4.5]	[3]
Operating Sys. firm		1:[1.4-8.5]	[3]
	5.4:1		[84]
Jet Propulsion Lab.	75%	$1:$100	[48]
Bell-Northern		1:[2-4]	[197]
Bull HN	80%	1:[1.43-6]	[233],[234]
Shell Research		1:30	[67]
IBM		1:[15-25]	[129]
Jet Propulsion Lab.		1:[10-34]	[128]
A Large Real-time software project		1:[6.3-11.6]	[56]
Ericsson	65%	1:16.75	[59]

Table 4.2: Reported peer review benefits in industry

motivate participants [93]. These problems could arise due to a previous failure with reviews, or wrong perceptions about reviews. Reviews are sometimes perceived as low-level manual work that can be easily automated and replaced by testing [54].

It is worth noting that most of the industry reports are coming from large software organizations like IBM [78] [127], Hewlett-Packard [96] [83], ICL [132], AT&T Bell Laboratories [16] and Ericsson [59]. Successes and failures of small to medium organizations with reviews are rarely reported in the reviews literature.

4.4 A framework for process taxonomy

Engineering decision making is a three-step process. First, plausible solutions for the target problem are proposed. Second, the feasibility of enacting each solution is assessed based on its risks, costs and benefits. For each alternative, risks are assessed to evaluate: *chance of correctness* and *chance of success*. Chance of success refers to the chance a proposed solution will adequately solve the target problem. Chance of correctness refers to the chance the proposed solution be deployed correctly. Based on the assessed risks, the expected benefits from enacting each alternative is calculated. Then, the costs associated with deploying each solution are factored in. The last step is to select the most appealing solution based on a predetermined criteria (e.g., lowest cost, maximum benefit).

Following these steps, a process engineer, when selecting a process alternative to solve a particular problem, answers the following questions in sequence:

1. What are the process alternatives that match this particular problem?
2. What are the chances that a selected process alternative will adequately solve the problem?
3. What are the chances that the organization can deploy each of the selected process alternatives correctly?
4. What is the expected benefit from each process alternative - taking into consideration the assessed risks?
5. What are the costs associated with each process alternative?

In order to answer these questions properly, knowledge about the processes along different dimensions is required. The economics of different processes (*economic dimension*), as well as the required development environment to support process deployment (e.g., required tool support, staff training) (*support dimension*) are key characteristics that need to be captured by any taxonomy supporting decision making in software process. In addition, a third dimension capturing the "how to" aspect of the process is needed (*technical dimension*).

The taxonomy introduced here is organized around three main dimensions *Technical, Economic and Support.* Each dimension encompasses a set of attributes required to identify that dimension. Hence, carefully planning for and being aware of these dimensions and attributes can help an organization to choose the best process alternative that suites their needs for a particular situation and thereby maximize the benefits within allocated costs.

In the next section, details of the dimensions of the taxonomy and the attributes characterizing them are discussed further in the context of peer reviews.

4.5 A taxonomy for peer reviews

In order to customize the process taxonomy presented earlier for peer reviews, we need to articulate fundamental concepts of reviews around the taxonomy dimensions. We elicited these concepts and notions from the literature and extended them as attributes and sub-attributes along each of the three dimensions. While the attributes are selected to be relevant to most processes deployed in software development, the sub-attributes principally apply to just peer reviews. Figure 4.1 shows the elicited concepts and presents them as sub-attributes for the three dimensions.

In the rest of this section, each of the dimension, attributes and sub-attributes are discussed. Reference to various research contributions and current industry practices are integrated into the discussion.

4.5.1 Technical dimension

The technical dimension of peer reviews is concerned with the strategies and enaction details of a particular review process. The main attributes of the technical dimensions are the **process objectives**, **process structure** and the **work product**. These attributes are discussed in the following sections.

Process Objectives

The set of process objectives is probably the most important characteristic that shapes the review process. For example, if an objective is defect detection, the preparation period becomes essential for review success. Input materials should include checklists. If, however, the objective is learning, the preparation

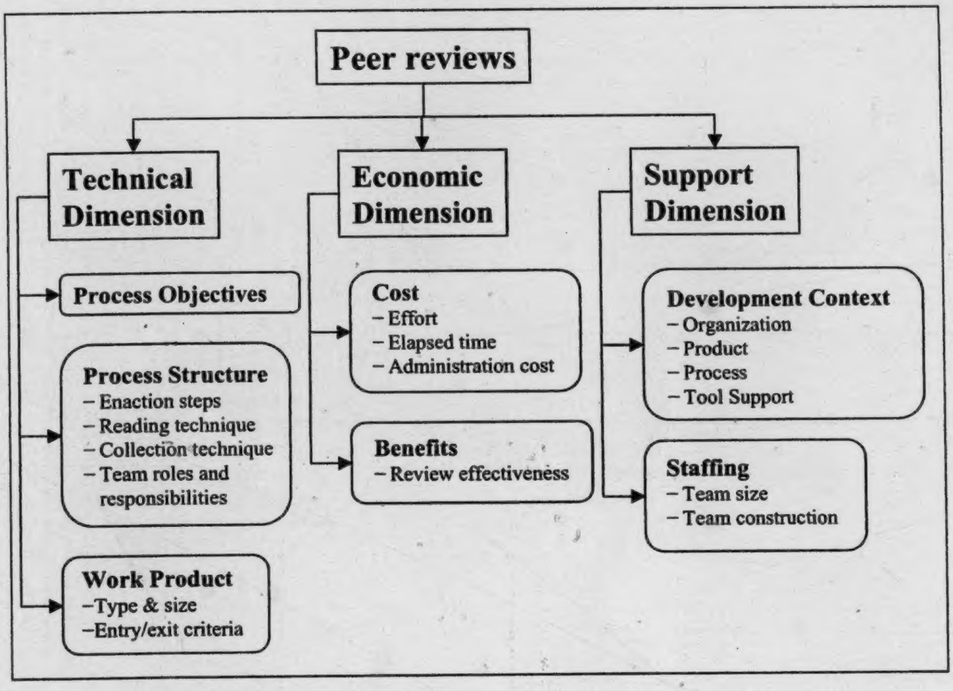

Peer reviews
Technical Dimension
Economic Dimension
Support Dimension
Process Objectives
Process Structure
– Enaction steps
– Reading technique
– Collection technique
– Team roles and responsibilities
Work Product
–Type & size
–Entry/exit criteria
Cost
– Effort
– Elapsed time
– Administration cost
Benefits
– Review effectiveness
Development Context
– Organization
– Product
– Process
– Tool Support
Staffing
– Team size
– Team construction

Figure 4.1: Dimensions and attributes of proposed taxonomy of peer reviews

stage can be small or even eliminated.

Software organizations have implemented different types of review processes to serve different objectives [219]. Reviews are mostly employed as a tool to verify and validate work products [42] and to identify, and subsequently fix defects in these products [78]. Nevertheless, information swapping and learning [110], as well as progress reporting [57], are also important purposes for deploying reviews. Some enactions of the review process have involved brainstorming sessions, or a forum for design decision. Other enactions have been geared towards resolving design and implementation issues [176].

Process structure

The process structure attribute details the overall organization of the process (i.e., enaction steps), specific techniques deployed during process enaction (e.g., reading techniques and collection techniques) and, resources mandated by the review process (e.g., team roles and responsibilities).

Enaction steps. This sub-attribute looks into the underlying process steps that characterize the review process. It helps in reducing ambiguities regarding how to conduct a particular review technique. A reference model is needed to explain the similarities and differences among different review processes. In general, the review process can be viewed as a series of phases:

Overview The overview phase consists of a meeting, usually referred to as the kick off meeting [92]. During the meeting, the author explains the work product under review to other review participants. The objective is to inform the participants about key aspects of the work product in order to speed up the understanding process.

Work product analysis Analyzing work products is an individual activity performed by each review participants. A major goal of this phase is for review participants to familiarize themselves with the work product under review [3] [34]. However, many researchers [230] [3] [90] [141], include defect detection as a major outcome of this phase. An objective that motivated further research to improve reading techniques during this phase [183] [18] [20] [141] [142].

Findings collection After review participants analyze the work product, their findings have to be collected and documented, then passed to the author to take proper actions. Findings collections is usually performed in a group meeting [77] [92]. During the collection phase decisions are made about what needs to be reworked and whether the work product needs to be reviewed again.

Rework. The final step in a typical review is to reflect the review findings on the work product being reviewed. The author needs either to resolve each raised issue or justify why the work product is defined the way it is [77] [208].

In main stream reviews literature, these phases are done in sequence. However, some authors [155] suggest multiple, parallel sessions of analysis and collection phases.

In parallel to the review phases, **administration** [180] is another performed activity; although it is rarely discussed in the literature. Before a review begins, administrative tasks include: selecting participants, preparing and distributing review materials and ensuring that the work product to be reviewed passes the entry criteria. During the review, the tasks focus on facilitating the review (e.g., scheduling meetings and if needed, tool support). After the review, the focus shifts to collecting and maintaining the review reports properly, ensuring that findings are handled properly by the author and checking the review exit criteria. Some organizations create the role of Chief Moderator [96] to administrate reviews and study possible improvements in the process.

Reading technique. During the preparation phase, different types of analysis methods are applied to work products. Reading is one of the key activities performed during this phase [20]. These methods range from unsystematic *ad hoc* methods [208] to highly systematic methods. Checklists and questionnaires are the most commonly used tools to help reviewers analyzing work products [77] [224].

The design of a checklist and/or questionnaire used on a review, should reflect the review objectives, reviewers' responsibilities, and the underlying work product analysis method [108]. Checklists for 'defect detection' might be passive, reminding reviewers about the issues they have to examine [53]; while checklists for 'correctness' may be more active, asking reviewers to justify the acceptance or rejection of a specific part of the document [207] [206]. A single general checklist can be designed for all reviewers. Alternatively, a different checklist for each reviewer can be designed to focus the reviewer's attention on a limited set of issues [164] or to match the reviewer's background and expertise [18].

Different approaches based on scenarios [20], reading by stepwise abstraction [71], and active participation from the document author [176] are also proposed and evaluated in practice.

The justification for research in reading techniques is based on the assumption that review results depend on the participants and their strategies for understanding the work product they are reviewing [182] [192]. However, when Sandahl *et al.* [198] replicated the experiment performed by Porter *et al.* [183] they concluded that the work product under review is the most probable explanation for the source of variance in defect detection rates rather than the reading technique as originally suggested by Porter *et al.* [183].

Collection technique. There are three basic techniques to collect reviewers' comments: group-focused through meetings, individual-centered and computer-mediated. Most of existing review processes include a meeting involving all review participants. It is believed that meetings produce synergy and participant stimulation and, as a result, better and more objective reviews [77] [219]. Learning by interaction is another advantage of a group-focused technique. However, scheduling overheads and the general problems of improperly conducted meetings have increased the popularity of the individual-centered techniques [66]. In these technique, meetings are held only when needed and attendance is optional [110]. The paper [230] suggests replacing meetings with depositions, in which the author and moderator meet with one reviewer at a time.

Although collection meetings are the most suggested collection technique in the literature [237] [84] [219], there are no conclusive results to support the effectiveness of such meetings for defect finding. In fact, some industrial data [59] indicate that they are extremely expensive. However, meetings provide other intangible benefits such as sharing development experiences, and enhancing team spirit [62]. Furthermore, collection meetings improve participants' confidence in the review quality [120].

In the third technique, called computer-mediated, a computer support environment is necessary. Several such environments have been developed [117] [156] to overcome the problems with group-focused technique without a serious risk of loosing its benefits. In this technique, meeting time constraints are relaxed, hence allowing participants to "effectively" meet [156]. Group e-mail and electronic communication [117] are depended upon to reduce significantly or eliminate the need for face-to-face meetings.

Team roles and responsibilities. A minimum set of roles for all reviews consists of [130]: moderator, author, reader and recorder. The author is primarily responsible for creating the work product, the moderator facilitates and coordinates the review, the reader guides the review session by reading or paraphrasing the work product, and the recorder keeps track, in writing, of the issues raised during the session. A chief moderator role [96] is created to oversee the administration of review enaction in the organization.

Work product under review

The work product attributes describe the nature of the work product that may undergo review (input material size and type) as well as the rules controlling the initiation, progress within the enaction steps and termination of the process (entry/exit criteria). Attributes of the technical dimension are furthered discussed in the following subsections.

Entry/exit criteria. Most review processes define entry and exit criteria that determine the starting or completion conditions for the review. For example, a clean compilation is usually the entry criteria for code documents [84]. With the exception of code documents, entry criteria is not well defined in the literature.

Exit criteria is usually dependent on the work product under review [119] and its properties [133] [134]. Some authors recommend the use of exit criteria based on statistically estimating the work product quality (number of remaining defects) [14] [50] [73]; evaluating these models is still an active research subject [74] and is explored further in chapter 6.

Input material type and size. Any software work product can, and in most cases should, be reviewed. Typical work products include: requirements, design specifications, code and test plans. Despite the fact that deferring defect detection is a costly mistake [35], code is the most reviewed work product [140].

The volume of materials to be reviewed depends on the review objectives and the required exhaustiveness of the review. Reviews focusing on progress reporting tend to cover a lot of material with shallow analysis, while defect detection reviews cover less materials with deep analysis. Materials that can undergo a review process can be work-in-progress or completed. However, an entry criteria for the work product is always recommended.

4.5.2 Economic dimension

The economic dimension describes the effect of the process on the project by detailing its **cost** and **benefit** attributes. It is essential to collect and maintain data that can be translated to either costs or benefits of the peer reviews.

At the micro level, cost-benefit data is essential to keep the project under control. At the macro level, the same data is used to evaluate peer reviews against alternative quality processes. For example, if the process goal is defect detection, a typical set of data to collect might be effort and number of defects found. At the micro level, this data is used to track total project costs, assess product quality, etc. At the macro level, the same data may be translated into cost per defect. This information can then be used to answer the question: if we did not find these defects during a review, how much more will it cost to detect and eliminate them later (e.g., using extra testing time).

Cost

For peer reviews, different types of costs are accounted for in the literature: direct cost associated with deploying peer reviews (effort and administrative costs) as well as indirect costs associated with the effect of reviews on the project schedule (elapsed time) [180] [140].

Effort and interval are probably the most important cost items for running reviews in large organizations; however, initiation, administration and maintenance costs are typical items that can affect the decision making process in small to medium size organizations.

Effort. Because peer review is a human intensive activity, the person-power cost (effort) of practitioners directly involved in the review process accounts for the major portion of total review costs. Effort is usually measured in person-hour units where typically, a person hour is rated equally despite that person's level of involvement in the project. However, this assumption is often not true in practice. Dollar value and a person's availability are two important factors directly affecting the value of his/her person-hour. For example, the value of the lead engineer's person-hour is typically higher than that of a junior developer.

Elapsed time. The elapsed time between start and end of the review process is called the review interval. The length of this interval depends on time spent performing the reviews and delays due to the unavailability of one or more of the review participant. Measuring review interval and subintervals is usually based on keeping track of start and finish times for the visible events of the review process [241]. The review interval time is an important metric because the inclusion of a review in the development process can directly increase the product's time-to-market. Increasing project time can lead to [180]: late market entry, opportunity costs and carrying costs. Accounting for these cost items falls beyond the scope of our work[3]; however, for proper calculation of these costs, delays attributed to reviews has to be measured and assessed. It is worth noting that mapping review interval to project delays is not a straightforward process. For example, if the review interval is one week, its effect on the project schedule could range from no effect to one week delay. The exact delay depends on whether the review is on the project critical path or not, the effect of tasks performed by review participants on the project schedule, etc.

Administration costs. Administration costs account for costs that stem from including reviews in the development process, for example, preparing the review material, maintaining review results, resources consumed during the review (e.g., meeting rooms), etc. all contribute to the overall costs of a review. Initiation costs account for costs related to starting up the process in the organization. Staff training, tool purchase and process adaptation are typical examples of initiation costs. Finally maintenance costs account for other running costs at the macro level, for example, tool upgrades, continual training, etc.

---

[3] For interested readers methods for calculating these costs falls under "cost accounting" domain boundaries. A good overview of these methods can be found in [151].

Benefits

Although software organizations can benefit from peer reviews in three different areas: defect detection and elimination, development process management [57] and enhancement of the proficiency level of review participants [110], most of the published work quantifying review benefits focus mainly on defect detection, (see for example [4] [169] [197] [67] [234] [16] [96]).

Review effectiveness. For defect detection purposes, several efficiency models have been used in the literature. Two straightforward efficiency models are suggested: *i*) measure the percentage of the total defects found during the review [77] ($\frac{\textit{defects found during review}}{\textit{total number of defects}}$) or *ii*) measure how the effort consumed in the process is made use of ($\frac{\textit{defects found during review}}{\textit{effort consumed in review}}$) [92], [150].

Another model suggested by Collofello and Woodfield [57] compared the estimated savings of the review to the costs consumed to perform the review ($\frac{\textit{estimated effort saved}}{\textit{effort consumed in review}}$). A similar formula is used in Hewlett-Packard to calculate the return on investment (ROI) obtained from the review [83]. In their formula they used the net savings instead of the estimated savings ($\frac{\textit{defects found during review}-\textit{effort consumed}}{\textit{effort consumed}}$).

Kusumoto [138] noticed a discrepancy in the above model. Comparing the savings to the effort consumed might be deceiving. For the same efficiency level, extra effort consumed is associated with larger savings. For example, suppose we are comparing two review techniques, the first one has an ROI of 10 and consumes 20 Hrs; the second has an ROI of 8 and consumes 50 Hrs. Looking only at the ROI, the first review should be favored. However, if we calculate the actual savings, the first review saved (10 x 20 = 200 Hrs.) and the second review saved (8 x 50 = 400 Hrs.). Kusumoto [138] suggested an alternative formula to compare the net savings to the total efforts consumed in testing, if no reviews are performed.

4.5.3 Support Dimension

The support dimension defines attributes characterizing the required support from the software organization in order to enact a successful peer review. Understanding these attributes facilitates comparison among alternatives while selecting the proper review for a particular project. Attributes of the support dimension capture the characteristics of the development environment surrounding the process through the **development context** and the **staffing** attributes.

Development context

The development context attribute articulates organizational culture (organization), nature of the developed product (product), process formality (process), and the required tools to support the review (tool support). With the exception of tools, few articles discuss this attribute of peer reviews (see for example [82], [208] and [96]).

Organization. Organizational context include the definition of the organization's business goals, resources and infrastructure, teams involved in the process, their agents and roles as well as information flow paths. Organizational context affects both the software process and product but in different ways. The effect on the product was recognized decades ago by Conway [60], who wrote, "Organizations which design systems are constrained to produce systems which are copies of the communication structures of these organizations." The effect of the software process and product was recognized in many international standards such as ISO 12207 [114].

Although an organization's operations and goals have, in general, broader scope than the software process, the software process is implemented in this context. At the macro level, to choose, or improve, a peer review process that effectively benefits the overall software process, the appropriate organizational environment needs to be established first [82]. At the micro level, the need to understand the organizational context by project managers is vital in order to choose the practices that both meet the organization's goals and integrate smoothly with its environment. For example, review practices and resources allocated for a project that produces the first in a family of products, will differ from those required for a mature product line.

Product. The nature of the application being developed or enhanced imposes different requirements on the development process. Enacting a tool or a method does not only depend on the process but also on the product. For example, the success of technical reviews for discussing the quality of a software design does not only depend on how rigorously the process is defined, but also on how clearly the system design is described.

Other factors include product size, maturity, level of reuse within the product, and the criticality of the product to the organization. The requirements for security, quality, maintainability, reliability, also constrain the process to be used.

Process. Incorporating changes in the development process in not an easy task. The maturity of the overall process in the organization defines the nature of tasks that can be successfully injected into or properly managed within the organization. Peer reviews are no exception. Developing a strategy for introducing reviews or changing current review practices is vital for the success of the process change program. Tools used to enact process improvement programs (e.g., training sessions [82], presentations [197], introduction of new automation tools) can be deployed here as well.

Measures that relate directly to the process capabilities of the software organization are the important parameters to consider along this sub-attribute. Assessment of the process maturity [68], defines how rigorously the process is defined and enacted within the organization. The importance of understanding the process capabilities of the organization emerges when applying reviews in practice, as things can easily go wrong [208] if the process is not well understood or consistently used.

For a particular peer review to be enacted successfully, its process requirements should not exceed the organization's process capability. For example, a highly formal review process (such as Fagan Inspection [77]) with well defined process objectives, step definitions, entry and exit criteria for each step and process monitoring mechanisms will not work properly for an organization with an overall *ad hoc.* approach to software development. Using a less formal review process (such as Freedman and Weinberg's Walkthrough [84]) might be more suitable for this organization.

Tool support. Tools to support reviews can range from tools as simple as checklists and findings-collection forms [84] to computer based support environments [119]. Computer based support for peer review is still in its infancy. Few tools are available for peer review support and they provide limited support for the process. Many tools start from the research community (e.g., Inspec$Q$ [134], [29] ASSIST [146] and HyperCode [178]) and others are commercially available (e.g., ReviewPro [191] and CheckMate [157]). In addition to these tools there is at least one "remote inspection" service available [189].

The level of support for these tools varies from support for a specific review process [134] to supporting limited phases of the process (e.g., findings collection [178]) to supporting the whole process [149].

Staffing

A paradigm reducing the importance of people in production was initiated with the industrial revolution through the creation of production lines. This is not true in software production, "*People are the production* when it comes to software" [15]. The best peer review for the project depends on both the technical skills of the project staff and the social environment in which they work. Proper training and harmonizing of the software worker's goals and motives are the key factors for a successful enaction of any peer review. The staffing attributes describe the team size as well as how the review teams are constructed.

Team size. Review processes reported in the literature have the number of review participants ranging from one reviewer [235] to an unlimited number [112]. Reviews with an unlimited number can involve tens or hundreds of participants who are usually not peers. These types of review are usually held by software contracting agencies to evaluate the progress of their contractors or by management to ensure progress. In this thesis we are not interested in reviews with unlimited participants and therefore do not consider them further in our work.

Fagan [77] recommended a four-person team. Performance of two-person teams has also been studied [34] [139] [181]. Typical numbers from industry ranges from three to eight participants [243] [238] [150] [234].

The review literature does not provide definitive rules of when to increase or limit the review team size. In a recent study, [38] supports adding more reviewers to increase the quality of the review. Usually, a trade off between work product coverage and cost must be considered in deciding actual team size. The smaller the team size, the more likely that certain findings will be overlooked. On the other hand, the larger the team size, the higher the impact on the project cost and schedule.

Team construction. Peer review is a human intensive process, hence, considering the human factor in the review team is fundamental for its success. In order to choose the review team properly, qualifications and motivations of the personnel have to be assessed. Qualifications include, general experience, experience with the current project, initial education and amount of ongoing training. Motivation includes the social aspect of the process, like career progression, salary, team work, etc.

To set the right attitude towards reviews, software developers have to accept the fact that defects in software are inevitable [83]. For defect finding reviews, reviewers have to be assured that review results are used to asses software particulars (e.g., quality, reliability) rather than the quality of the work product author [78]. Hence, it is recommended to exclude management from these type of reviews [83], [128] [216]. Furthermore, 'selling' reviews to project participants through advertisement campaigns is deemed useful to resolve any wrong impressions or attitudes toward reviews [197].

Primary candidates for the review team are the project participants [78]; they are the ones with most knowledge about the work product under review. However, reviewers with distinguished expertise may be invited [216] [215]. It is always recommended to have well trained [3] [82] inspectors with good experience and knowledge about the work product under review [78] [219]. For purposes like team building and knowledge spreading, novice or less expe[illegible]ence reviewers should also be included. Other suggested participants include maintenance experts, and user representatives [216].

4.6 Summary

This chapter presented a taxonomy framework to distinguish several software processes with the same objectives. The framework organizes the process along three dimensions: technical, economic and support. In addition to the process objectives, the technical dimension is concerned with the methodological details of the process: process structure and work product characteristics. The economic dimension is concerned with the cost effectiveness of the process by describing its cost drivers and benefit measures. Finally the support dimension is concerned with the environment within which the process is enacted. The framework was applied on peer review processes and successfully identified 16 different proposed and practiced review processes which were organized in an experience base for peer reviews - see appendix B for a summary of the different review processes identified.

The taxonomy presents an up-to-date overview and analysis of peer review knowledge presented in literature that assisted in identifying 17 different process attributes. Process objectives, enaction steps, reading techniques, collection techniques, team roles and responsibilities, work product type and size, entry/exit criteria along the technical dimension; effort, elapsed-time, administration costs and review effectiveness along the economics dimension; and characteristics of the development context (organization, product, process, tool support) and team size and team construction along the support dimension. These attributes characterize the core concepts of the review process, allowing the identification of the details of the review process that best suite a particular situation through the comparison of different alternatives along various dimensions.

Chapter 5

Enacting the Competency Refinery

5.1 Introduction

Associated with most software development environments is a lot of experiences residing in people. It is often difficult to manage experiences within organizations [30] because they are many and varied. Without proper management; however, organizations may easily waste time and effort collecting and managing experiences [200]. The question is how can useful experiences be identified, collected and disseminated to those who need it? Experience collection activities must impose minimal overheads while collecting experiences and, ensure completeness and consistency while disseminating these experiences. In order to assure these qualities in an experience management system, it is necessary to enact the system. In particular, lessons learned from process enactment are needed to develop and recommend processes supporting experience management activities properly.

In this chapter, we report on a study of processes supporting the infusion of new technologies in a software development organization. The study was conducted over a two-year period from 1998 to 2000 in an educational setting. The main goal of the study was to investigate processes and techniques supporting software development using application frameworks. Specifically, we investigated how software developers with little or no knowledge of a framework approach the development of new applications using this framework. This chapter focuses on issues related to experience communication, elicitation and preservation.

5.1.1 Study objectives

The focus of this enaction of the competency refinery is to observe and understand issues related to framework usability from the framework users' standpoint. The refinery's objective is to solicit related experiences and ultimately, recommend best practices. In order not to disturb the project logistics (e.g., budget, schedule) by the refinery processes, we believed that the best approach was to observe practitioners developing application with minimum interaction. We solicited the experiences by reviewing project deliverables, meeting with project participants regularly and polling their perspective using questionnaires. By examining the information collected in a postmortem fashion, we gained insights about the principles and issues. However, we provided guidance when needed and performed a postmortem analysis for the projects we studied.

Since the enaction took place in a classroom environment, we felt it is necessary to give the student some guidance, including what we envisioned as best practices and guidelines. Using these recommended practices was optional and students were allowed to make use of their expertise and experiences to modify these guidelines. We observed the processes enacted by the students and determined the usefulness of our guidelines. Early on, we narrowed down the scope of the refinery to these questions:

1. Can peer reviews be effectively deployed as an interaction mechanism to communicate knowledge about the framework at the early stages of application development?
2. Is there a pattern-of-use framework users deploy to understand frameworks and learn how to develop applications using frameworks?
3. What are the roles of framework documentation in communicating the framework knowledge and how can it be improved?

To answer these questions, we collected a wide range of data about the participating students, factors affecting framework usage and understanding, and the students' views on recommended practices.

5.2 Background

Frameworks are difficult to understand [37]. This difficulty creates learning problems associated with building applications by extending frameworks. The static and dynamic structure of the framework must be first understood before adapting it to the specific requirements of the application. Understanding these structures requires a lot of time and effort from new users of the framework [185], [186]. Some researchers estimate the required effort as being equivalent to that of maintaining an existing application [210]. Problems associated with understanding a framework usually hinder the development process, and in extreme cases may cause projects to fail [37].

According to the breakdown of the learning processes suggested by Nonaka and Takeuchi [172], learning how to extend frameworks is a two step process: knowledge externalization and knowledge internalization. During knowledge externalization, framework developers explicitly codify all knowledge required to extend the framework. During knowledge internalization, framework users need to internalize the provided information into their tacit knowledge. A survey of the framework publications shows that little work has been done to support knowledge internalization in frameworks as opposed to knowledge externalization.

Knowledge externalization in frameworks focuses on strategies for framework developers to document their work. The basic concept promoted by most of the work addressing the framework intended-use is to think about the framework in terms of the functionality it provides. Proposed strategies suggested the use of hooks [87], patterns [121], motifs [143] or cookbooks [137] to document the functionality in terms of sets of related functions. For example, a hook describes a set of functions supported by the framework along with a demonstration of how to extend the framework to provide this functionality to an application. Other strategies demonstrate the most important points of the framework functionality using exemplars [89] or tutorials [229].

All framework documentation strategies assume that framework developers will be able to anticipate future uses of the framework and provide enough documentation for all these uses. This assumptions imposes two main deficiencies:

1. It is hard to anticipate how much knowledge is enough for the framework users to use the framework effectively.
2. It is unlikely that the framework users will always extend the framework in a manner that was conceived by the framework developers.

To overcome these deficiencies and support the documentation, the framework users need to participate in the process of understanding the framework by asking questions in order to emphasize their perspective.

Documentation strategies provide no support for knowledge internalization. The assumption is that framework users will internalize the knowledge by reading. We believe that supporting knowledge internalization is as important as supporting framework documentation in order to facilitate framework learning.

We selected peer reviews as a vehicle for knowledge communication due to its unique educational capabilities [118], [67]. Fortunately, reviews encompass most of other knowledge communication techniques used within the software engineering domain (e.g., reading, lectures and, learning by active participation). They support the exchange of views about a framework with the objective of solving usage problems through an organized process. Another viewpoint about the educational capabilities of reviews, endorsed by Votta [230], argues that education by observation and participation is not effective, and that proper training courses are a better option. This is a viewpoint that is questioned by other studies that showed that lectures, the most common method in training courses, are the least effective in knowledge transfer and that learning by active participation is more effective [220].

5.3 Refinery context

We conducted the study over three consecutive terms within the context of a senior level software engineering course (CMPUT401) at the Department of Computing Science, University of Alberta. The study focused on a group project that was the major activity and the focus of the course. The pedagogical goal of the project is to help students put the theoretical knowledge acquired through their undergraduate program into practice and show their ability to work in teams and communicate verbally and in writing with external and internal interest groups.

Students were divided into teams of five to seven students each. In order not to interfere with team synergy, students self-selected their team partners. To compensate for any bias in the collected data that might result from the self-selection process (e.g., teams are not of equal capability), each student's background information (e.g., courses taken, industrial experience, technical knowledge) was collected and considered during data analysis. A total of fifteen teams participated in the study, six in the first term, five in second term and four in third term.

Over the time span of the course, each team was required to develop a small to medium size client-server application of their choice. The only requirement imposed on the product was that it must be built by extending framework called the Client Server Framework (CSF), which is described in the next section.

The development budget ranged from 90 to 100 person-hours per student. All developed applications were written in Java with a communication component that used the Internet as a communication medium. The size of the developed applications ranged from 21 to 108 classes (excluding the framework) covering a variety of application domains like on-line auctions, distant learning, on-line reservation systems, document sharing systems and on-line gaming.

5.3.1 Study participants

Data was collected on all of 89 students who participated in the study. All subjects were computer science students in their senior year of which 53% had at least sixteen-month industrial experience through an industrial internship program.[1] Using students as representatives of software professionals is a common, but still controversial, practice in software engineering studies [147], [210]. A replicated experiment [184] indicates that student subjects can be adequately used as representative for software professionals. Because, more than half of our subjects have some industrial experience, we believe our results are relevant to industry practice.

5.3.2 Organizational Structure

The projects are designed to simulate how real software products are developed. The realistic setup was primarily achieved by following the domain engineering organizational model [39]. In this model, a domain engineering unit is responsible for the development, evolution and support of the reusable asset (the CSF framework). Products are developed in separate business units. The projects were run as business units and a member of the teaching team (the CSF expert) represented the domain[2] engineering unit by providing the support activities for the CSF framework.

The teaching team performed the role of upper management, overseeing the development activities and establishing a quasi-corporate culture through coordination mechanisms. The projects were monitored by upper management through weekly meetings and were controlled by a set of pre-scheduled deliverables. Commitment ethics [109] were adopted to ensure a mature attitude among team members. The issues emphasized within the course were:

- Requirements are negotiated between members of the development team and upper management.

---

[1] http://www.cs.ualberta.ca/iip/index.html

[2] The domain in this case represents the domain of services provided by the framework rather than the application domain.

- Agreement on what is to be done was identified in a product specification document.
- Teams are to do their best to meet their commitment,
- If it is evident that the delivery can not be done before the commitment date, advanced notice is given to upper management and a new, less ambitious, commitment is negotiated.

In addition to upper management activities, I, as a member of the teaching team, played the role of the process adapter to fine tune the review process to satisfy the project needs and track the process enaction.

5.3.3 Development process

Due to the lack of guidelines for framework deployment in the literature, we decided to develop our own. We relied heavily on experience existing from previous offerings of the course, and the experience base we developed for peer reviews. We separated the project life-cycle conceptually into two phases: exploring the framework to gain an understanding of its use and, using the framework to build an application.

In the first phase, framework users explored the basic functionality of the framework. The objectives of this phase were:

1. To ensure that the framework can adequately support the needs of the application the team was to develop.
2. To understand the framework enough to build a reuse strategy (e.g., assigning development responsibilities to team members).

At the beginning of the first phase, the CSF expert, gave two ninety-minute overview sessions of the framework. The sessions covered the framework design and its documentation style. The use of the framework was also demonstrated using a simple example application to give a concrete instance of the abstract classes of the framework. The developers were given the chance to explore and voice their concerns about the framework in a peer review session. This review, the focus of this study, aimed at supporting the internalization process of the framework knowledge.

In the second phase, the framework was extended to produce the application. The main objective of this phase was to submit the project deliverables on time. Details of the process were left to the individual project teams to decide upon; however, we provided guidelines as to the nature of a set of deliverables at predetermined milestones. Each team had to produce an analysis document and a detailed design document. Deliverables up to and including product testing, consisted of updated version of the two documents along with test plans, integration plans, reports on the process used, and user documentation.

A second technical review was held in the eighth week of the development cycle. The objective of the second review was for the teaching team to review the product design and give students some feedback. The second set of reviews followed a process similar to the first set, but they differed in roles, and reading technique. In the second review, the teaching team was the reviewer and student team was the authors. No particular reading technique was suggested for this review.

Project progress was monitored in two ways. Weekly meetings were held between management and each project team to gauge their progress and address any concerns the team might have. Secondly, project team members were required to keep time logs of their project-related activities.

5.4 Projects context

Building the application by extending the CSF was the only development constraint imposed on the products developed within the context of this study. In addition, two review processes were recommended as part of the development process. All fifteen teams chose to perform the second review, and twelve of them chose to perform the first one (the focus of this study). In this section we describe in more details the CSF framework and the process details of the first review.

5.4.1 The CSF framework

The CSF [85] is a small framework of approximately 50 Java classes developed to serve the purpose of this study. The framework facilitates persistence data management and platform-independent communication. Through a relatively simple messaging mechanism, the framework allowed objects within different programs running on different machines to exchange messages of any type and size.

In order to facilitate its use, the framework comes with several types of documentation covering all aspects outlined in [49] and [123]:

- Design documentation to provide a high-level overview of the major classes of the framework and their relationships to one another. This includes both class diagrams and collaboration diagrams along with textual descriptions.
- Hooks [87] to document the framework's intended use. They show how and where the framework can be extended in order to meet application specific requirements.
- Use-cases to give an overview of the use of the framework. They refer to individual hooks where developers have to provide their own classes or methods.
- Examples to show some specific uses of the framework and to provide running code that the developers can experiment with.
- Interface descriptions and source code to show details of classes used in the framework.

The documentation is about 25 pages (excluding code and code documentation) and was distributed using the Web. The framework was carefully designed. Commonality analysis was performed on other existing frameworks in the area along with other client server applications developed in the class. Design patterns [88] were incorporated where applicable. Furthermore, a beta version of the framework had been used in a limited manner in two student projects of a previous offering of the course to ensure the maturity of the framework.

5.4.2 Peer review for information sharing

The objective of the first review in the project was to facilitate the knowledge internalization step associated with learning the framework. During the review we collected questions and identified difficulties in understanding the framework. The questions were addressed immediately if possible, or later if needed. The underlying assumption is that framework understanding would be enhanced by deploying a peer review process. The success of this process mandates that all framework documentation to be reviewed must be in a relatively mature status.

The review team consisted of a moderator, framework expert, recorder and five to seven reviewers (all members of the development team). Members of the teaching team were assigned the moderator, the framework expert, and the recorder roles. I joined the review meetings as an observer and was responsible for tracking the review progress, capturing relevant data and video taping all review meetings.

Unfortunately, none of the existing review processes included in our experience base address the learning objectives directly. The round robin reviews proposed by Freedman and Weinberg [84] was the closest to serving our purpose (see Appendix B), as round robin reviews provide a good educational environment, especially when all reviewers are at the same level of expertise [84]. In this study, we enacted a process based on the round robin review. The review consisted of three phases: preparation, consolidation and follow-up. The phases are detailed as follows:

Preparation Phase The reviewers were exposed to the review materials in two phases. First, they had a preparation meeting (1.5 hour in lecture format) with the framework author. During that meeting, the framework author introduced the framework, its design principles and available documentation. These materials were available to students two weeks before the review meeting. On the second phase, students uncovered and recorded their findings using a checklist based method [183]. One week before the review meeting, checklists were sent to students. Students individually reviewed the framework documentation and each student prepared a list of questions to ask.

Consolidation Phase Review findings were consolidated in face-to-face collection meetings. Each team was given the opportunity in a thirty-minute meeting to pose their concerns and/or questions. Depending on the nature of the concern/question the framework expert either discussed it immediately during the review meeting or deferred the answer to be published in a list of Frequently Asked Questions (FAQ). The moderator restricted the discussion about the raised issues as suggested by Gilb & Graham [92].

Follow-Up Phase The author follow-up method was used as described in [84]. The framework expert was free to choose what to include in the FAQ. The FAQ was published (or updated) one week after the review.

The preparation phase of the reviews started at the beginning of the second week of the project. The review meeting was held at the third week and the process ended by publishing (or modifying) the FAQ in the fourth week.

An inspection rate of 50 pages per hour was required in this study - which is much higher than the suggested optimal rate of one page per hour [92]. This may indicate an unusually ineffective review process; however, the situation is not as unusual for the following reasons:

- Gilb & Graham indicated that it is very difficult for organizations to achieve the optimal rate in practice. In practice, inspection rate of up to 60 pages per hour are reported (see for example [203], [40]).
- Studies investigating optimal rates are always referring to code inspection rather than documentation reviews. The Reviews literature rarely discusses technical documentation.

To support the review process, we developed preparation forms, collection forms and a participant information package (see Appendix C). During the review preparation period, participants used the preparation forms to record questions and concerns they wanted to raise. They also recorded preparation period, team ID, and personal ID. In addition to the material to be reviewed, the participant information package included the checklist developed to support analysis of the documentation. Findings forms were used to summarize the outcome of the review meeting.

5.5 Data and analysis

5.5.1 Data and analysis technique

A wide variety of data was collected over the course of the project life span, using several techniques. First we describe the subset of data related to this thesis, then we briefly discuss the analysis technique used to answer the research questions.

Questionnaires. Questionnaires were used at the beginning to collect information about students' academic backgrounds and industrial experiences, and at the end to poll students opinions about the provided process guidelines and the quality of the documentation. Students were also required to report on the effort invested in activities related to the framework uses. After the first term, as a result of the projects post-mortem analysis a second questionnaire was added to poll the participants' view about aspects of the framework at the end of the first phase of development.

Data on the first questionnaire were mostly confirmed using students' academic records. Information from the final questionnaire were partially verified against the submitted project documentation and the weekly reports.

Implementation score An implementation score was assigned for each product at the end of the semester. A grade was assigned by the teaching team based on the quality of the product's architecture and how well the submitted system meets each of the agreed-upon functionalities, as defined in the product specification document.

The data was collected over three consecutive terms and members of the teaching team changed over these terms. In order not to jeopardize the consistency of the data, we analyzed data from each term separately.

Review data The preparation data was collected (time and list of findings) and the review meetings were video taped. Details of the review data and its analysis will be discussed in the next chapter.

Self-assessments Within the term, students were asked to assess the effectiveness of all team members, individually and collectively. Self-assessment reports were used to detect problems, or exceptional performance within the team.

Process Management Report After committing to the product requirements, each team is required to submit a project plan. The project plan details the product requirements, development life-cycle, role assignment and the projected budget. At the final delivery, the project plan was modified to reflect changes to the plan over the semester with proper justification in case of deviation from the original plan.

Progress reports Each team was requested to update their project logs on weekly basis to record development activities done during that week as well as the number of hours worked by the team on the project.

Problem reports Clarification or help with the CSF was submitted via e-mail to the framework expert.

To answer our first question (can peer reviews be effectively deployed as an interaction mechanism to communicate knowledge about the framework at the early stages of application development?), we statistically analyzed the review data (number of findings and effort). Statistical findings were further confirmed (or questioned) based on the subjective assessment of the review participants. The reviewer's perception was collected right after the meeting in a quick discussion session and was confirmed using the questionnaires.

To answer the second question (is there a pattern-of-use framework users deploy to understand frameworks and learn how to develop applications using frameworks?), we qualitatively analyzed the project management reports looking for commonalities and differences in the development process. Specifically we were interested in management models and role assignments. Findings were confirmed using the self-assessment reports, progress reports and the observations we collected during the weekly meetings.

The third question (what are the roles of framework documentation in communicating the framework knowledge and how can it be improved?) was answered by quantitatively analyzing the student's response to some closed questions in the questionnaires. For example, students were asked to rate the usefulness of the examples supplied in the framework documentation in a five point scale. To reduce subject bias in the results, we allowed "not applicable" as one of the choices. Findings were confirmed using open ended questions in the questionnaires. For example, students were asked to recommend changes in the process for future offerings of the course.

5.5.2 Potential confounding factors

Since in this study we didn't have the same level of control as in a laboratory experimental study, identifying and eliminating the effect of potential confounding factors were vital for the validation of the study findings. The effect of subjects' backgrounds is a usual concern in this type of study. The concern is that the differences in the statistical data can be explained by the participants' backgrounds rather than the variables identified in the study. Typically, professional training of a software developer and industrial experience [210] are used to assess his/her background.

The amount of professional training was assessed in terms of academic records in previously studied computer science courses and the number of these courses. Based on their Grade Point Average (GPA), students were categorized as above average, average or below average. Assuming a normal distribution for GPA, a student is considered:

- above average if his GPA $\geq$ average GPA + $\sigma/2$,
- below average if her GPA $\leq$ average GPA - $\sigma/2$,
- average otherwise.

All subjects in our study have experiences in designing and implementing course projects, and they all worked in development teams. We also assessed their development experience in industrial setting. A student was assessed as:

- novice if s/he has any industry experience but less than one year of experience in industrial setup
- experienced if s/he has more than three years of experience in industrial setup, and
- limited experience otherwise.

Background data was assessed using a Likert-type scale in the following manner: above-average (3), average (2), below-average (1), experienced (3), limited experience (2) and novice (1). The team score is calculated as a percentage of the maximum score they could have achieved. For example, if the academic records of a team of six students shows two above average students, three average and one below average, the team score would be $\frac{2*3+3*2+1*1}{3*6}\% = 72.22\%$. Table 5.1, provides a summary of background profile of all participating teams along with the implementation score. The average number of computer science courses (count) is also provided in the table.

In order to understand the effect of the confounding factors, we used Pearson correlation coefficient [122] to measures the strength of the linear relationship between the implementation score and each of the potential confounding factors discussed above. Correlation r falls between -1 and $+1$. Values of r near zero indicate a very weak linear relationship. The strength of the relationship increases as r moves away from 0 towards either -1 or $+1$. The case of -1 indicates an inverse relationship. The values of Pearson correlation

Team ID	Industrial Experience	Professional Training		Implementation Score
		Score	Count	
T1G1	53.33%	60.00%	6.00	98.67%
T1G2	50.00%	72.22%	4.67	92.00%
T1G3	38.89%	72.22%	3.17	88.67%
T1G4	40.00%	60.00%	4.20	55.33%
T1G5	66.67%	83.33%	6.83	93.33%
T1G6	66.67%	80.00%	7.80	83.33%
T2G1	38.89%	61.11%	4.50	88.81%
T2G2	60.00%	73.33%	6.80	80.00%
T2G3	61.11%	61.11%	6.00	90.71%
T2G4	44.44%	55.56%	5.67	16.67%
T2G5	61.11%	66.67%	6.50	75.24%
T3G1	33.33%	71.43%	3.71	85.71%
T3G2	52.38%	80.95%	6.57	76.19%
T3G3	33.33%	55.56%	4.33	84.52%
T3G4	66.67%	71.43%	7.43	92.38%

Table 5.1: Background scores for project teams

	Correlation with		
	GPA	number of courses	Industrial Experience
Term 1	0.1001	0.2571	0.3105
Term 2	0.3198	-0.1	0.1768
Term 3	-0.1026	0.2	0.4045

Table 5.2: Correlation between implementation score and student background

coefficients are summarized in Table 5.2. As seen in the table, there is some correlation between the confounding factors and the implementation marks. As expected, the industrial experience has the largest correlation coefficient. However, we can ignore these correlation in further analysis as the largest correlation coefficient, 0.4045 accounts for 16.4% ($r^2 = 0.1636$) of the observed variation in the implementation score. According to Humphrey [108]: there is no relation if $r^2 \leq 0.5$. Furthermore, the combined effect (over the three terms) of the industrial experience reduces the effect to 7.25% ($r^2 = 0.0727$).

5.6 Results

In this section the results are summarized in terms of project teams which is the same unit of analysis used in the project. Through our study, we used the implementation score as the major indicator for team success. The significance of this measure was cross-checked using records of the weekly meetings and the type and volume of questions addressed to the CSF expert after the first review.

5.6.1 Effectiveness of peer reviews

Our quantitative analysis confirmed the usefulness of peer reviews in the context of understanding frameworks (see Chapter 6). The subjective assessment of those who replied to the question "how helpful was the review process for improving your understanding of the framework?" in the questionnaire confirmed the statistical results. The students answers to this question are summarized in Table 5.3. As can be seen from the table, on a scale of 1 to 5 with 1 = not helpful and 5 = very helpful, 54.29% of those responded to this question rated the review as either helpful or very helpful as opposed to those who rated it to be with limited or no help 17.14%. One student even added *"The group discussion, and review led to a faster and more thorough understanding of the framework."* Furthermore, some of the reasons given by students for the low rating of helpfulness were: *"my understanding was that we were supposed to understand the CSF when we showed up, enough to raise risks and concerns with it"* and *"I don't think I am technically sound enough to go around and start commenting on other peoples code."*

Reviewers pointed out the benefits of reviews in three areas: *i*) setting deadlines for the understanding process, *ii*) consolidating the development team's point of view through well-organized discussion, and *iii*) getting fast feedback from the framework expert. Few reviewers saw reviews as waste of time because they suspected the accuracy of the documentation; as one student stated: "*(the project has to work with the CSF not the concepts explained in the documentation).*"

Helpfulness level				
1	2	3	4	5
0.0%	17.14%	28.57%	42.86%	11.43%

Table 5.3: Answers to question: "how helpful were the reviews in understanding the CSF?"

The review benefit extended beyond supporting the development organization, it provided an interaction mechanism to improve the form and content of the framework documentation. The information sharing reviews were used in building an experience base to support the CSF in the form of Frequently Asked Questions. In total, the FAQ contains 48 questions, 85.42% of which came directly from reviews.

Reviewers also detected framework defects (e.g., *"There seems to be unnecessary dependency between the persistence and the communication subsystems. The initialization requires a persistence manager to be initialized even if the user doesn't require persistence (just communication)"*) or documentation limitations (e.g., *"The exception types (i.e. SendException) do not appear to be listed in a definitive format"*). Reported defects amounts to 18.5% of the total number of reported issues.[3]

5.6.2 Role of the documentation

At the beginning of the study, to ensure that the framework documentation fulfils all the roles proposed by Johnson & Foote [123], we included a complete set of documentation as suggested by Butler *et al.* [49]. In addition to the functionality provided by the framework, we found that the framework users were interested to learn about the non-functional aspects of the framework as well. 25% of the reported questions and concerns in the reviews addressed the performance of the framework. Although it seems natural enough for developers to have concerns about framework performance, it has received little recognition as a documentation priority [49]. Some of the questions in this category addressed the general performance of the framework, (e.g., *"In*

---

[3] excluding false positives

Term	HL Doc.		D. Doc.		Code		Exp. Base		Consulting	
	R	C	R	C	R	C	R	C	R	C
1	2.1	n/a	2.7	n/a	4.0	n/a	n/a	~~n/a~~	n/a	n/a
2	3.4	3.4	3.8	3.2	4.2	3.4	3.6	3.7	4.5	4.4
3	2.2	3.1	3.9	4.0	4.0	3.0	3.6	4.0	3.4	3.8

Table 5.4: Rating for different sources of knowledge

real time context, what are the overheads imposed by the framework?") and others addressed the options provided by the framework, (e.g., *"Are there performance advantages to running "applet mailserver" vs. "mailserver"?*)

As part of the post project survey, participants were asked to rate on a scale of 1 to 5 where 5 is the highest value, the usefulness[4] of different parts of the supplied documentation. The results shown in Table 5.4 are the averages of these ratings categorized into high level documentation (design diagrams and use cases) and detailed documentation (hooks and examples) and code.

In the first term, as can be seen from the first line of the table, all aspect of the supplied knowledge was not perceived as useful by project participants (with the exception of code). Analyzing the results we realized that the documentation suffers from two main problems: presentation and perspective.

The problem with the presentation of the documentation appeared in the comment of one participant: *"The documentation was fairly sparse, and I really only found the use cases of any value."* Although the documentation provided all the required knowledge proposed in the framework literature, we did not adequately support navigation among different type of documentation. In the second term, we added diagrams and hyper-links to cross reference related parts of the documentation. In general, these changes improved how the developers valued the documentation as seen in the second and third lines of Table 5.4.

The second problem with the documentation was already known to us, the documentation reflects the perspective of the framework developer, providing what s/he envisioned as important to document rather than what is seen as important from the perspective of the framework users. The FAQ we maintained

---

[4] Starting the second term the usefulness was refined into relevance and clarity.

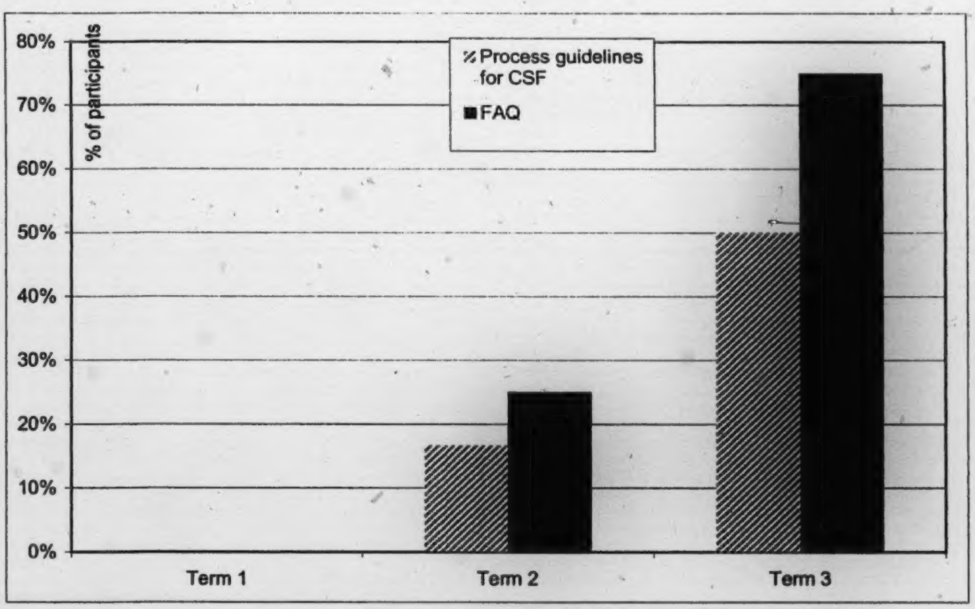


Figure 5.1: Percentage of students relying on the framework experience base to understand the CSF

was the first proposed solution to mitigate this problem. We also realized the need to provide process guidelines for using the CSF framework. A series of steps defining high level processes guiding new users of the framework through its use were provided starting the second semester of the study [86]. Students rated the experience base (FAQ) part of documentation as of high value (see Table 5.4). Furthermore, as the framework FAQ and the process guidelines evolved, the number of the students who relied on the framework FAQ for understanding and using the framework increased. This increase showed up in the increase of the number of students who explicitly mentioned FAQ and the process guidelines while answering the question: "Which option[5] or combination of options from question 3 did you mainly rely on in understanding the CSF?" in the post project survey. The results are displayed in Figure 5.1.

---

[5] The options are code, hooks, examples, design diagrams, use cases, process guidelines and FAQ.

5.6.3 Effective processes

The effect of using the framework on the development process was relatively uniform across all teams. During the analysis phase, teams investigated the framework to understand its architecture and how to integrate it within their application. For the first two term, the key process enacted during this stage was the information sharing review. During the third term the teams relied predominantly on the knowledge carried over between terms. To prepare for the review, students relied on high level documentation, mainly the use cases and design diagrams, to gain a high level understanding about the framework.

During the coding and testing phases, most of the developers relied on the code and examples. The key process during this phase was expanding the examples. Almost all project teams started from one of the supplied executable examples and modified that example to accommodate the requirements of their application. The experience base built around the framework (FAQ and process guidelines) provided guidance through out the modification process.

The number of developers directly interacting with the framework varied depending on the team management model used. The number of developers responsible for learning and using the framework varied from one person to the nearly the entire group. Our data shows in Table 5.5 that successful teams delegated all framework-related tasks to few team members. Pearson correlation coefficient between the implementation score (measure of success) and the number of people involved with the framework ranged from $r = -0.99$ to $r = -0.73$.

This correlation could be explained by considering a team's resource management. Developers involved with the framework have to invest a considerable amount of time and effort to understand the framework before they could become a useful resource for the team. Teams with few developers assigned to the job, gained collectively from having an expert with good understanding of both the framework and the application. This expert was able to work as a framework consultant and to propose solutions and answer question when the need arose without having other developers to devote extra time to under-

Team ID	# of developers involved with the CSF	Implementation Score
T1G1	1	98.67%
T1G2	2	92.00%
T1G3	3	88.67%
T1G4	4	55.33%
T1G5	3	93.33%
T1G6	3	83.33%
T2G1	3	88.81%
T2G2	2	80.00%
T2G3	1	90.71%
T2G4	1	16.67%
T2G5	5	75.24%
T3G1	1	85.71%
T3G2	2	76.19%
T3G3	1	84.52%
T3G4	n/a	92.38%

Table 5.5: Background scores for project teams

standing the details of the framework. On the other hand, teams with more developers assigned to the job depleted their resources repeating the same job with minimal gain to the team as a whole.

The use of consultant-based management was quite evident in the third term of the study, as one team even relied on the help of a student who had used the framework in a previous term. This model was recommended in other studies concerned with communicating knowledge in software organizations [116].

5.7 Summary

We conducted a study to investigate processes and techniques supporting software development using application frameworks. Specifically, we investigated how software developers with little or no knowledge of a framework approach the development of new applications using this framework. The study was considered successful. Over the course of three academic terms, adopted techniques managed to successfully disseminated knowledge about the framework to interested parties. The experience base centered around the framework significantly improved the framework documentation and drastically reduced dependency on the framework expert.

Analysis of quantitative data showed that reviews are useful interaction mechanism for information sharing at early stages of development (after the requirements phase and before the design phase). These findings were further confirmed by the subjective assessment of the study participants. Reviewers pointed out the benefit of reviews in three different areas: (*i*) setting deadlines for the understanding process, (*ii*) consolidating the development team point of view through well-organized discussion, and (*iii*) getting fast feedback from the framework expert.

Chapter 6

Packaging Process Experiences

6.1 Introduction

The focus of this chapter is to analyze the concrete experiences collected from the enaction of the competency refinery and to build a modus experience package for the information sharing reviews discussed in the pervious chapter. According to the review taxonomy defined in Chapter 4, the new package has to define the process objective, work products that can be reviewed, methods to calculate costs and benefits and the development context and to recommend an optimal team size and a process structure.

The main objective of the information exchange review is to support framework learning by speeding up the rate of learning. However, the reviews have proven to be useful in improving the quality of the framework documentation as well. In the pervious chapter we discussed the development context, and the document under review. In this chapter we define the costs and benefits of enacting peer reviews to accelerate the early stages of framework learning. We then use this function to statistically evaluate different alternatives of the process structure. Specifically, we want to answer the following questions:

- Does the checklist used affect the review results?
- What is the optimal size for the review team to maximize the review effectiveness?
- Do multiple-session reviews outperform one-session reviews with respect

to defect detection? If yes, what is the optimal number of sessions to maximize the defect detection rate?

First, our investigation strategy is presented in Section 6.2, followed by an overview of the data used in the analysis in Section 6.3. The research questions are answered in Section 6.4 and the effect of different process inputs on the review performance is assessed. Specifically, professional training and industrial experience of the reviewers, and the length of the preparation period are analyzed in Section 6.5. The chapter concludes with our recommendation for this type review.

6.2 Investigation strategy

A total of 11 teams (55 reviewers) participated in the reviews; 27 students from six teams, in the first term, and 28 from five teams in the second term. Since we want to answer our research questions for a varying number of team sizes, we based our study on virtual teams; a technique widely used in software engineering research [18], [38], [41], [163]. A virtual team is created by randomly selecting a combination of individual reviewers from reviewers working on the same document. The validity of this technique is assured by creating a large group of virtual teams for each team size and using the statistical parameters of the group as basis for the results.

The team size was systematically changed to cover the range of teams sizes required to properly answer that particular question. For each team size a group of 1000 different virtual teams were created. For each virtual team, we collected the effort in minutes of each individual during preparation (rounded to the nearest 10 minute), and the specific issues (findings) raised during the review meeting by that individual. To reduce the effect of out-liars in the data, the value of the variable of interest is reported as the median of the 1000 observation. For the same reason, the IQR (Inter Quartile Range) was used as a measure for the spread in data.

In the analysis, we considered two distributions to be significantly different only if the Student's t-test reject the null hypothesis (H_0) that the observations are drawn from the same population with a confidence level ≥ 0.9. In all cases, the value of the significant probability (P_t) is reported to indicate the strength of evidence against (or for) H_0. $P_t \leq 0.1$ indicates the significant probability for rejecting the null hypothesis, a smaller value for P_t means that the H_0 is strongly rejected or the result is highly statistically significant.

6.2.1 Evaluation criteria

For defect detection purposes, several models have been used in literature to report review effectiveness. Two straightforward efficiency models were suggested: either to measure the percentage of the total defects found during the review ($\frac{\textit{defects found during review}}{\textit{total number of defects}}$) [77], or to measure the results (defects found against the effort consumed in the process ($\frac{\textit{defects found during review}}{\textit{effort consumed in review}}$) [92] [163]. With the exception of evaluating the optimal team size, the first model was used in all the analysis.

6.3 Overview of data

Three sets of data are used in the analysis of this study:

i) *Preparation reports.* The preparation forms were used by individual reviewers to record questions and concerns they wanted to raise during the review meeting. They were also used to record time invested in preparing for the review. The data recorded in the preparations forms are used to assess the gain (or losses) of the virtual review meetings, assuming that all questions and concerns reported in the preparation forms will be asked during the virtual review meeting.

ii) *Meeting summaries.* Collection forms were used to summarize the review meetings. They captured all questions and concerns raised by each team during the meeting. This information is used to assess the benefits of multiple session reviews by comparing the meeting reports from all review sessions.

iii) *The author repair list.* The author repair list appeared in the form of a list of Frequently Asked Questions (FAQ). Each item in the FAQ list characterizes an issue related to the framework performance and functionality, but not covered by other forms of documentation (omission defects).

6.3.1 Data reduction

Data manipulation after collection is frequently called data reduction. This manipulation is usually done to remove data that are not pertinent to the study or to adjust to any systematic errors in the measurements.

The purpose of the data reduction we performed was to consolidate the information captured in the preparation and meeting forms. Because the questions and concerned were documented in natural language, the same issue may be worded differently in different forms. Following the data reduction technique presented in [41], two researchers independently reviewed the list of issues with a primary focus of identifying repeated issues. To ensure a consistent counting scheme, the two researchers then met to produce a consolidated list of unique issues. To raise our confidence in the final list, this list was further confirmed by having the framework author review and agree to its content.

Issues raised during the review meeting were classified and their classification was agreed upon by the reviewers. This classification was further validated with the framework author through an interview and a few adjustments were made. The issues were grouped into:

- False Positives (questions that are not related to the framework (e.g, Java related questions).
- Maintenance issues (issues for which documentation was changed to fix framework defects and documentation limitations (e.g., missing Web link).
- Findings (questions related to framework understanding).

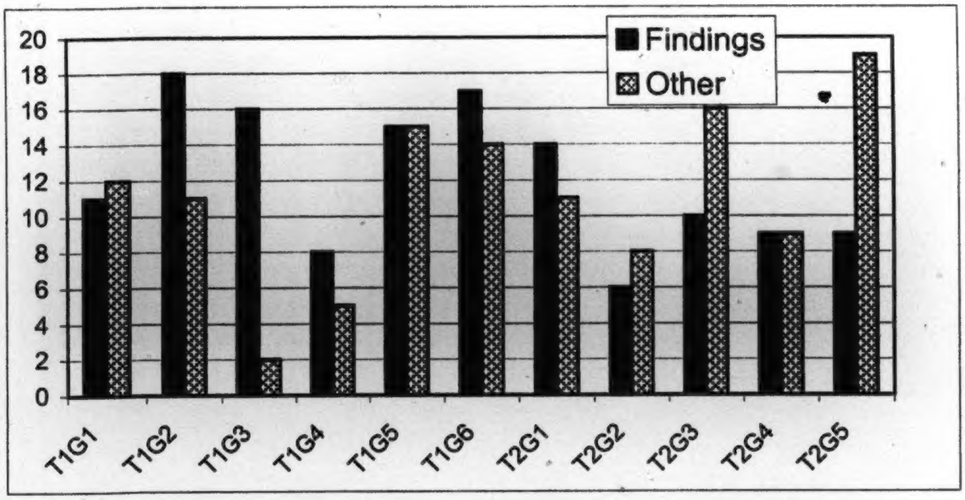


Figure 6.1: Disposition of issues recorded at the review meetings

Finally, to raise our confidence in the list of findings, it was cross-checked with the FAQs. All issues in the findings list either appeared in the FAQs or was resolved during the review meeting.[1] Across all reviews, 36% of the issues raised during the review meetings were false positives, 12% dealt with maintenance issues and 52% were issues related to framework understandability. False positives and maintenance issues were not included further in the analysis. The distribution of issues reported by each team appears in Figure 6.1.

6.3.2 Summary of observations

During preparation, reviewers examined the framework documentation to identify difficulties in understanding the framework. During the meeting, the issues raised were answered immediately, if possible, or later in the FAQ. The meetings served two functions: (1) Removal of unimportant or unrelated issues from the list of questions that has to be answered in the FAQ, and (2) improvement of the framework expert's understanding of the issues. In this section, we analyze the discovery of findings across the review activities.

---

[1] Issues resolved during the meeting generally resulted in documentation changes (e.g., changes in the documentation wording to make it clearer).

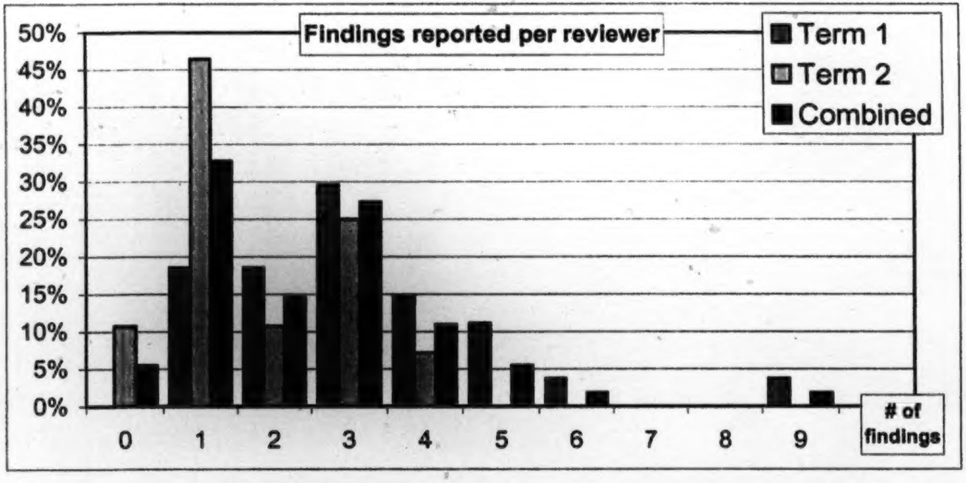


Figure 6.2: Number of findings per reviewer during preparation

Summary of reported findings

Figure 6.2, shows a histogram[2] of the number of findings reported by each reviewer. The number of recorded findings across all reviewers range from 0 to 9, with a median = 2, IQR = 2.

In analyzing the rate of findings reported per term, we found no statistical difference[3] between the average number of findings reported across teams in each term (p_F(Term 1)= 0.37, p_F(Term 2)= 0.62). However, this average was significantly different across terms ($p_F = 0.01$). The rate of reporting per team in Term 2 was 45.6% less that the average rate in the first term.

Summary of effort data

The total number of hours spent in preparing, meeting and responding to the review findings is the most common measure for review cost [77], [57], [138]. Figure 6.3 shows a histogram of the time investment by each reviewer in order to prepare for the review. Across all reviewers the effort ranged from 30 minutes to 4 hours, with a median effort was 2.5 hour, with IQR = 1 Hour.

---

[2] All histograms in this thesis are normalized to show percentage with respect to the total population size.

[3] Multiple comparison tests like these were computed using ANOVA F-test.

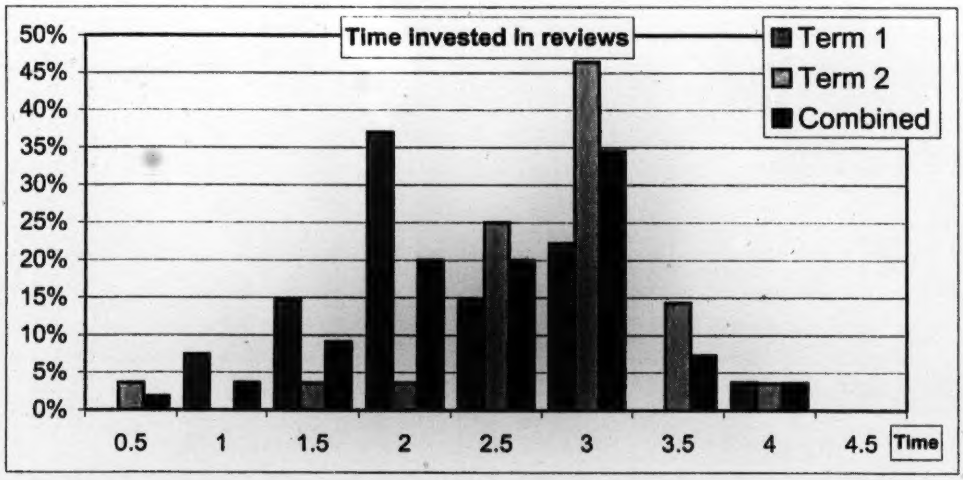


Figure 6.3: Time spent preparing for the review

The data suggests that there is no statistical difference between the average effort across teams ($\rho_F = 0.33$).

6.4 The effect of process structure

6.4.1 The effect of the detection technique

The choice of the detection technique has always been perceived as an important factor affecting peer review performance [183]. Some methods use systematic techniques, with specific and distinct responsibilities (e.g., Active design review [175], phased inspection [133]). While others use nonsystematic techniques with general and identical responsibilities (e.g., Fagan inspection [77], *N*-fold inspection [155]).

In our study we adopted the nonsystematic approach, however, we supported the reviewers with a checklist. In the first term, two different checklists were prepared and each list was given to three review teams. The first was a generic checklist based on the checklist provided with the description of the round-robin reviews [84]. The second checklist was based on the scenario based checklists proposed by Proter et al. [183].

We studied the effect of using different checklists in two stages. First,

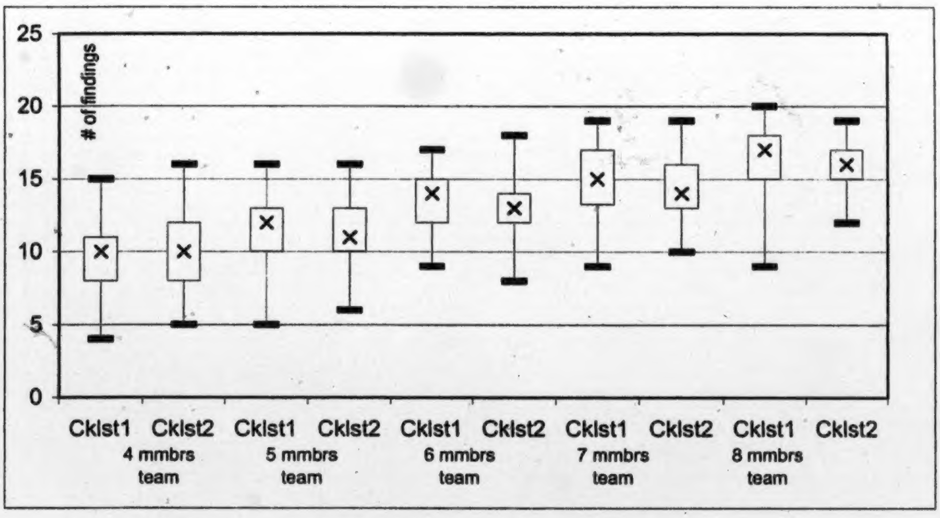


Figure 6.4: The effect of using different checklists

we investigated the individual preparation reports. There was no significant difference in the number of findings reported by users of the two checklist ($p_t = 0.57$).

Second, we investigated the amount of overlap in the reported findings reported by the reviewers. In order to perform this analysis, we created virtual teams. The total number of findings reported by each virtual team was then calculated and we analyzed the differences in these results. The team size was systematically changed from four to eight reviewers. The range of team sizes was selected based on literature recommendations [95], [78], [150], and typical numbers from industry [238], [243], [234].

As can be seen in the boxplot in Figure 6.4, there was no difference between the two checklists despite the team size (ρ_t (4 members team) = 0.43, ρ_t (5 members team) = 0.73, ρ_t (6 members team) = 0.80, ρ_t (7 members team) = 0.97, ρ_t (8 members team) = 0.70).

6.4.2 Large team versus small team

The nature of the review and the definition of effectiveness are the major source of variation in recommended team size. Usually, a trade off between work-product coverage and review cost must be considered in deciding the actual team size. The smaller the team size, the more likely that some findings will be overlooked. On the other hand, the larger the team size, the higher the impact on the project's cost and schedule. In our context, the effectiveness of the review is calculated as the gain divided by the costs. The review costs are calculated as the sum of fixed costs and the cost of performing the review. The fixed costs is the sum of the administration costs and the cost of the framework expert. The review cost is the sum of the time reviewers spend in preparation and the cost of attending the meeting. The review cost can be formulated as:

$$\mathit{Fixed\ costs} + \mathit{Team\ size} * \mathit{Meeting\ duration} + \sum_{i=1}^{\mathit{team\ Size}} \mathit{Preparation\ time}$$

Review gain is calculated as the sum of the individual gain of all review participants. The individual's gain is the time saved due to attending the review meeting, i.e the number of of issues a reviewer learns during the meeting multiplied by the cost of detecting these issues by reading the framework documentation. The review gain can be formulated as:

$$\sum_{i=1}^{\mathit{team\ Size}} \left(\mathit{Issues\ learned\ by\ reviewer}(i) * \frac{\mathit{Preparation\ time\ of\ reviewer}(i)}{\mathit{Issues\ detected\ by\ reviewer}(\mathrm{i})} \right)$$

This formula is built on assuming that the cost of detecting the issues learned during the review is uniform and equals to the cost of detecting an issue during preparation time. These assumptions are not accurate.

Since we don't know the exact cost for an individual to detect more issues we simulated this value. The time consumed by a reviewer to find an issue after the review was estimated as the time consumed to detect an issue during the review multiplied by a multiplication factor (MP). We simulated the MP by a normally distributed random variable. A low MP means that issues discussed during the review are simple and the reviewers will discover them if they invested a small amount of extra time reading the documentation. The higher the MP, the higher the likelihood a reviewer may need more time to detect that particular issue by reading the documentation.

Since the review focused on the reviewers initial understanding of the framework, the preparation time was divided between understanding the basics of the framework and studying how the framework can be extended to

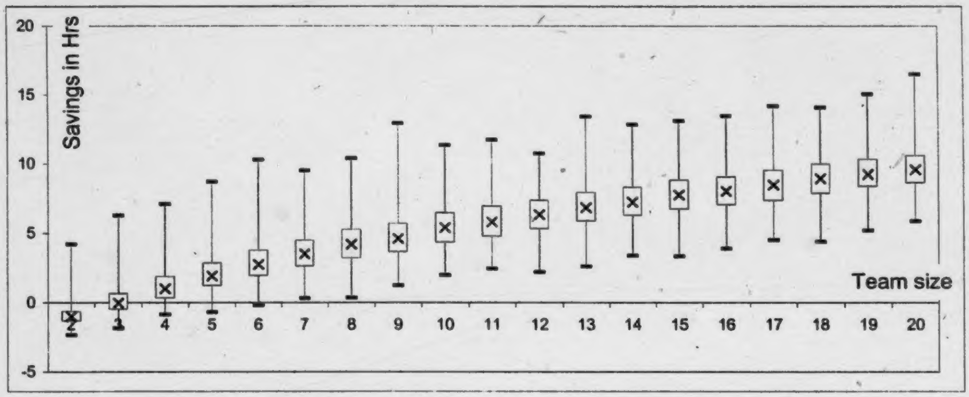


Figure 6.5: Review benefits as team size increases

meet the project requirements. During the studying time, concerns and questions are raised. We assume that a considerable part of the preparation time was dedicated to understanding the basics of the framework. Consequently, it is highly probable that finding more issues will take less time. Hence, we assumed that the value for the MP should range between [0,1]. In the analysis we used MP = $(\mu = 0.5, \sigma = 0.167)$[4].

Although reviews with more than eight reviewers are rare in practice, one report supported review teams of twelve reviewers [73]. Increasing the number beyond 12 was supported by a recent study [38]. In order to cover all sizes of review teams mentioned in the literature, we changed the team size from two[5] reviewers to teams of up to twenty reviewers. Because the framework documentation were changed after the first term, members of the virtual review teams were selected from the same term.

As can be seen in the boxplot in Figure 6.5, adding more reviewers will increase in review benefits at a higher rate than the increase in its costs. Hence, increasing the team size will increase the collective benefit from the review.

For practical reasons, while determining the optimal team size, we considered a bigger team to be more efficient only if adding a new reviewer will result in cost saving of more than 30 minutes (0.5 hrs.) per reviewers. The goal is

---

[4] Following the six-sigma rule, 98% of the values are bound between [0,1]

[5] for team size 2 the maximum number of different teams that could be created equals $^{27}C_2 = 351$ virtual teams.

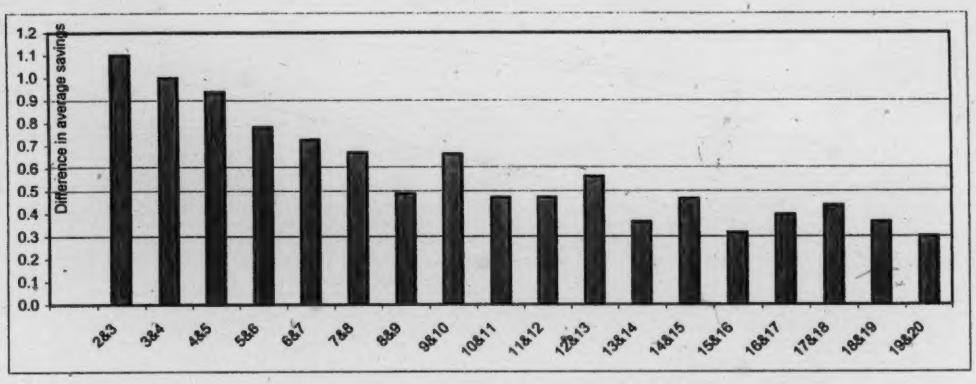


Figure 6.6: Difference in review benefits as team size increases

to evaluate the number beyond which, adding more reviewers will not achieve the predefined savings level. According to this definition, the optimal team sizes are nine or ten reviewers (as seen in Figure 6.6). The average increase in time saving per reviewer drops to 25 minutes per reviewer when the team size increases from 10 to 11 reviewers. Note that we considered the average savings as the team increases from 8 to 9 reviewers (29 minutes.) as an outliar. On the other hand, the data indicates that the benefit per reviewer for two and three person reviewers is less than 30 minutes as well (see Figure 6.5).

6.4.3 One session versus multiple sessions

From the document under review perspective, the findings that appeared in the FAQs are omission defects [18]. Finding and correcting these defects are as important as spreading the framework knowledge among students.

As seen in Section 6.4.2, under realistic assumptions, large teams are not efficient with respect to the reviewers' knowledge about the framework. Increasing the number of review teams [155] was proposed as an efficient method to increase the throughput of the process; especially when the document under review is written in natural language (e.g., requirements document). In this section we want to evaluate the effectiveness and efficiency of using more than one team to review the same document.

In order to perform this evaluation we generated all different two-team, three-team, four-team and five-team permutations from our data such that all

	One	Two	Three	Four	Five
Term 1	6	${}^6C_2 = 15$	${}^6C_3 = 20$	${}^6C_4 = 15$	${}^6C_5 = 6$
Term 2	5	${}^5C_2 = 10$	${}^5C_3 = 10$	${}^5C_4 = 5$	${}^5C_5 = 1$
Total	11	25	30	20	7

Table 6.1: Multiple session - data summary

teams are from the same term. Table 6.1 shows the number of data points for each treatment. Furthermore, to analyze the multiple session data, we performed another data set reduction to remove duplicated findings (i.e., each finding is considered once in the findings count across sessions).

We calculated the average change in the review throughput as

$$\frac{\textit{average findings for } N+1 \textit{ sessions} - \textit{average findings for } N \textit{ sessions}}{\textit{average findings for } N \textit{ sessions}}$$

As can be seen in Figure 6.7, adding an extra session to the review has improved the review throughput by a minimum of 14.8% (from four-sessions review to five-sessions review) to a maximum of 80% (from one-session review to two-sessions review).

Despite the difference in exact values, these results are consistent with the results reported by Martin & Tsai [155] and Schneider *et al.* [199] but contradict the results reported by Porter *et al.* [181]. This difference can be attributed to the type of document under review. In our case, as well as in Martin & Tsai and Schneider's *et al.* experiments, the documents under review are written in natural language. The document under review in the experiment performed by Porter *et al.* was more formal as reviewers reviewed code written in C++.

6.5 The effect of the process inputs

Several factors, other than process construction, may affect the throughput of the review process, including the characteristics of the document under review, reviewer's ability to detect issues, and the framework expert. The number and type of issues raised during preparation are influenced by the reviewers's ability to detect and raise questions as well as the clarity of the document under

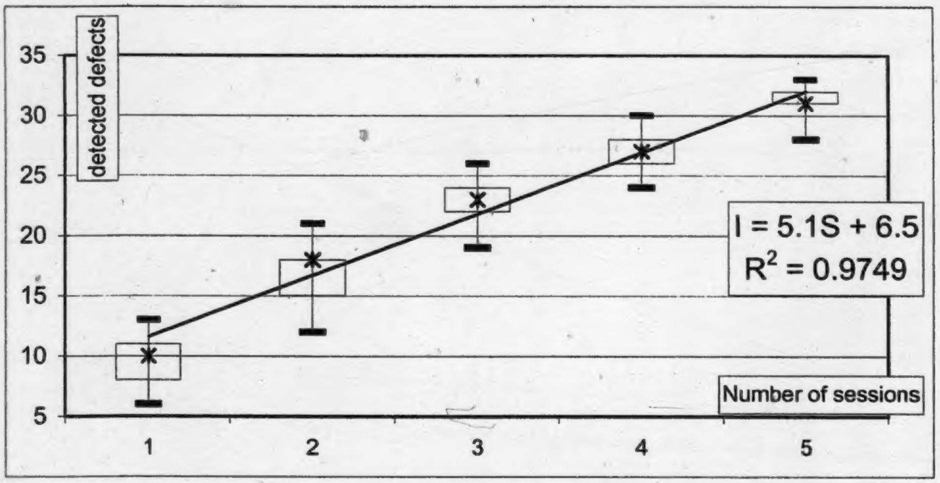


Figure 6.7: The effect of multiple session on review results

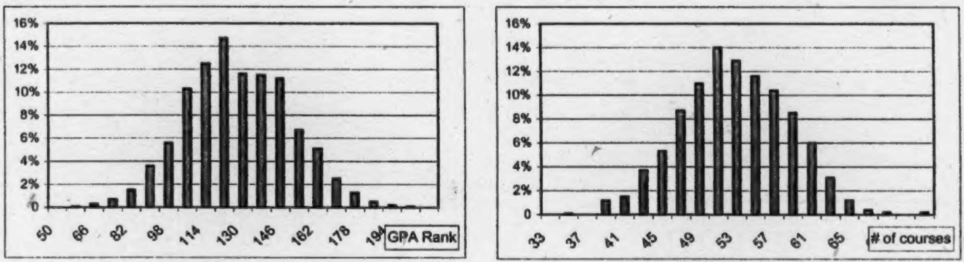
review. The number and types of issues recorded in the collection meeting are influenced by the number of issues recorded in preparation and the amount of discussion about a specific issue with the framework expert attending the review meeting.

In the previous section we studied the effect of the process structure on the review process. In this section we discuss the reviewer's effect on the review throughput. Specifically, we will assess the effect of the following three factors: professional training, industrial experience and preparation time on the total number of issues reported by each reviewer.

Although, it does not confirm causality, the effect of these factors is assessed by calculating the correlation between the number of reported findings and each of the three factors. In order to increase the confidence in our results, we used resampling techniques [211] to calculate the correlations.

6.5.1 Professional training factor

It is an established fact that experienced practitioners outperform less experienced ones. Apart from the hands on experience (discussed in the next section), process improvement programs like CMM put a lot of emphasis on professional training as a vehicle to enhance practitioners' experience. In this



(a) (b)

Figure 6.8: Histogram of randomly selected academic information

section we want to evaluate the effect of professional training on the review throughput.

Although we did not provide direct training on reviews, we believe that students' Grade Point Average (GPA) of junior and senior level computer science courses as well as the number of these courses can be taken as an indicator to the effect of professional training on the review performance. However, the results have to be interpreted cautiously and can not be generalized beyond the performance on this type of reviews.

In order to evaluate the correlation between the GPA and the review performance, we investigated whether students with high GPA tend to report a high number of findings more often than would be expected by chance. The strategy is to rank the students by GPA and split the number of reported findings into 'above median' and 'below median'. Then, the sum of ranks (SOR) for students in the 'above median' category is calculated ($\text{SOR}_{\text{observed}}$).

By randomly associating the GPA rank with the number of reported findings and calculating the resulting SOR, a data point representing an association 'by chance' between the sum of ranks and the 'above median' category is generated. By repeating this process 1000 times, a distribution representing how often the SOR are associated, by chance, with reporting 'above median' number of findings is generated (see Figure 6.8(a)).

The level of correlation can be evaluated by comparing the $\text{SOR}_{\text{observed}}$ to the random pattern of SOR. The more frequently that the randomly generated SOR(s) is as low as the $\text{SOR}_{\text{observed}}$, the higher the probability that there is no

Number of Reported Findings	Industrial Experience Rank		Total
	2	1	
Above Median	25.9%	7.4%	33.3%
Below Median	37.1%	29.6%	66.7%
Total	63.0%	37.0%	100%

Table 6.2: Data summary for industrial experience versus reported findings

relationship between the students GPA and the number of reported findings.

The $\mathrm{SOR}_{\mathrm{observed}}$ for the students reporting an 'above median' number of findings is 115. Comparing with the randomly generated values we find that in only 36.3% of the trials did random selection of ranks produce a total of 115 or less.

Following the same analysis strategy, the observed number of junior and senior level computer science courses taken by students reporting 'above median' number of findings sums to 47 courses. Comparing with the randomly generated values (see Figure 6.8(b)) we find that in 84.7% of the trials did random selection of number of courses sums to 47 or more.

The above results imply that the association between the review performance and the GPA is stronger than that with the number of courses. However, in both cases it does not seem to be a strong enough association to be used as a predictor for the review performance.

6.5.2 Industrial experience factor

In this section we statistically evaluate whether the level of industrial experience affects the individual's performance in the review or not. According to the categorization of industrial experience data presented in the previous chapter, student's experience can take the value of 1 for less than one year of industrial experience or 2 for more than one year of industrial experience.

If there is a high association between the two variables, then the observed data will be large in the two diagonal cells on either diagonal on Table 6.2 (ignoring the 'total' data). Under the assumption that industrial experience and the number of reported findings are positively associated, we expect that the sum of the top-left, bottom-right diagonal to be larger than the other

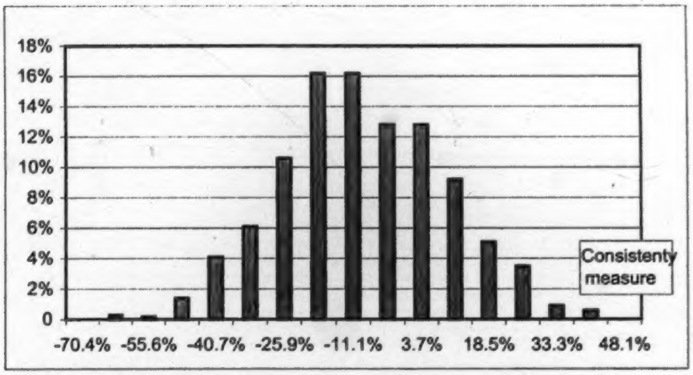


Figure 6.9: Histogram of consistent industrial data through random selection

diagonal. The difference between the sums of the two diagonals is called the consistency measure. In Table 6.2, the consistency measure = (25.9% + 29.6%) - (7.4% + 37.1%) = 11%)

Comparing the difference between the sums of the two diagonals in the observed data with randomly generated simulated results we found that 10.1% of the trials produced difference equivalent to or higher than the observed difference (see Figure 6.9). The results indicate that a strong association exists between the reviewer's industrial experience and the number of reported findings.

6.5.3 Preparation time factor

The amount of preparation time is a measure of the amount of effort the reviewers put into studying the framework documentation. In this study, we recommended that students spend between 120 to 180 minutes in preparation. However, students did not follow this recommendation literally, the reported preparation time varied from 30 to 240 minutes.

Ranked on the number of reported findings, the top 25% students spent on average 168 minutes in preparation for the review. The average time reduces to 160 minutes for the top 50% students. As can be seen in Figure 6.10, the chance of detecting the same numbers in a randomly generated data is 2.9% and 4% respectively, indicating a high correlation between the amount of time invested in preparation and the number of defects reported during the review

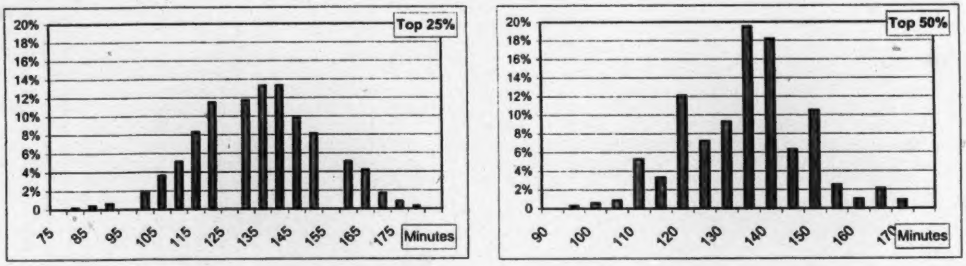


Figure 6.10: Histogram of randomly selected effort data

meeting.

However, this impressive correlation can only be used as an indicator for the review performance rather than a predictor. The amount of effort invested does not only depend on the 'planned' or 'recommended' amount of effort the reviewer wanted to spend in preparation, but also on the document under review as well. It is influenced by the number of questions and concerns a reviewer has identified by reading the documentation. The more difficulties s/he finds, the more time s/he will invest to formulate the questions and concerns. Hence, recommending longer preparation time may not improve the review throughput. Comparing average preparation time and reported defects among teams in the first term and the second term confirms this fact. Although there is no significant difference in the preparation time among the two sets, reported defects in the second term is less by 45.6%.

6.6 Summary and recommendations

We have collected data from a review process designed to accelerate the framework learning process. In this study, reviews are used in a novel way where benefits of the review are not only directed towards the reviewers but focus on improving the quality of the document under review. In order to build an experience package for this type of review, we evaluated the optimal team size, number of review sessions and, reading techniques. We also evaluated the effect of the review context, in particular the effect of the reviewer's background (industrial and professional training) as well as the amount of time invested

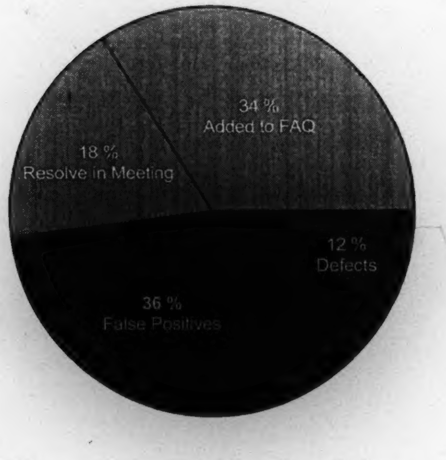


Figure 6.11: Findings breakdown

in preparing for the review.

In the following section we summarize the specific results of our study data and discuss their implications from the viewpoint of a practitioner wanting to deploy the experience package and a researcher working to improve the experience package.

Reported Findings. Around one third of the reported issues turned out to be false positives. Approximately, 10% of the issues raised maintenance issues either in the framework or in the documentation. Finally, of the issues related to framework understanding questions, two thirds were seen as omissions from the documentation. The remaining 18% were resolved during the review meeting (see Figure 6.11). The number of issues related to the framework understanding per review team ranged from 25.3% to 56.8% for the first term and from 18.9% to 44.2% for the second term .

For practitioners this suggests that time spent in the review is well invested; on average 38% of the issues raised during a review meeting added to the knowledge of the reviewers on how to use the framework. We anticipate that this number would increase if the reviewers are familiar with the programming language used to develop the framework, as most of our false positives were questions related to the Java programming language. However, the decline in the number of questions and concerns discussed in the review meeting from the first to the second term suggests that the review process is more useful at the introduction of a new framework. Over time, the documentation should evolve to cover most of the concerns and questions related to the framework with respect to that domain of applications.

For researchers this suggest that the learning capabilities of reviews need to be well understood to support different aspects of software development involving program understanding. Better models need to be developed to evaluate specific aspects of the review in this context (e.g., review efficiency and meeting gain).

Team size. We found that the bigger the team size, the more issues will be detected. However, if the cost associated with adding a new member to the team is accounted for, the optimal team size will depend on the difficulty level of the issues raised during the review. Under reasonable assumptions, we found that the optimal team size is 7 or 8 reviewers.

For practitioners, this imply that increasing the team size up to 8 would improve the throughput of the review. However, for smaller development teams, we don't recommend joining two teams in one review as the points of interest in the framework may change from one project to another (e.g., not all the teams used the persistence storage aspect of the framework).

This result was based on the preparation reports. The underlying assumption is that all issues raised during the meeting are understood by all reviewers attending the meeting. More research is needed to verify these results by taking meeting synergy into account.

Multiple sessions. We found that increasing the number of sessions improves the throughput of the review; however, the rate of improvement drops drastically as the number of sessions increases. Two sessions reviews improved the throughput over one session review by an average of 80%. The improvement drops to 28% if the number of sessions increased from two to three. In practice, this means that if the main objective of the reviews is to improve the quality of a framework documentation, multiple session reviews improves the review throughput.

The efficiency of multiple session reviews depends on the number of defects to be found in the document. The lower the number of defects in the document, the less the number of review sessions needed. In order to design the proper number of sessions for a review, more research is needed using statistical techniques to estimate the number of remaining defects in a document.

Reviewer's Background Our observed data indicates that the performance of individual reviewers is highly correlated with their industrial experience. A weaker correlation with their performance in computer science courses was also observed. The number of courses taken did not seem to have any correlation with the performance.

From a practitioners' perspective, these results confirm the value of industrial experience in software engineering activities. The importance of professional training is also evident but somewhat weaker.

6.6.1 Summary

In summary, an information sharing review is an effective tool to accelerate the process of knowledge internalization for object-oriented application frameworks at the early stages of the development process. The optimal team size for this type of review is 7 or 8 reviewers. Statistically, the design of the checklist does not affect the review performance; however reviewers felt more comfortable with the scenario based checklists. In general, experienced reviewers will outperform the less experienced ones; however, professional training may mitigate this performance difference.

Chapter 7

Automated Support for the Competency Refinery

7.1 Introduction

A typical competency refinery has to deal with a large amount of information in order to create business value to support the development organization. The refinery built in this thesis is no exception. The amount of information to be handled mandates some level of automated support for the refinery functionality. The main goal of the support environment is to facilitate the reuse of past experiences and thereby suggest solutions to given problems based on some similarity criteria.

The refinery's support environment should aid both the users and the maintainers of the experience base. The most important users are project members who must be provided with easy to use and efficient search capabilities of the experience base. Key support for the maintainers of the experience base include the capturing and analyzing of information about the process enactment.

In the following section we discuss requirements for the proposed support environment. An overview of the tool architecture to support different aspects of the refinery architecture are described in Section 7.3. The documentation of the prototype tool we built along with some usage scenarios follows in Sections 7.4 and 7.5, respectively. An assessment of the developed prototype along with suggested modifications to overcome identified problems is discussed in Section 7.6. Finally, Section 7.7 summarized the chapter's contributions.

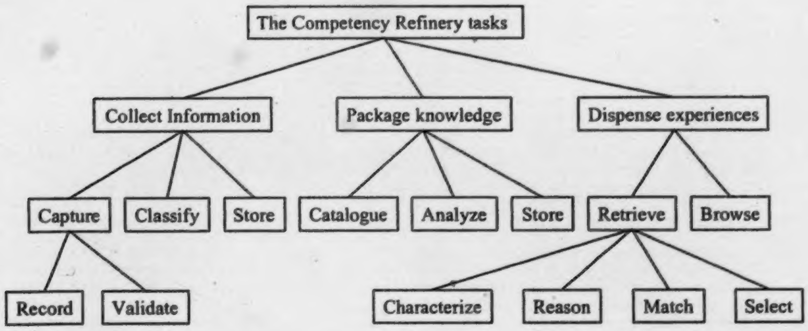
7.2 Requirements for supporting the Competency Refinery

Several authors [7] [26] [24] [102] [167] [196] [221] [240] discuss the important factors required for a successful implementation of a system to support knowledge management activities within software organizations. These requirements can be grouped into organizational requirements (e.g., motivating employees, need to share knowledge and to create shared objectives), functional (e.g., experience acquisition, experience characterization) and technological requirements (e.g., details of the knowledge base and other technical infrastructure). Although the organizational requirements are extremely important, we will not elaborate on them any further, as they require substantial psychological and sociological studies that are beyond the scope of this thesis.

7.2.1 Functional requirements.

As discussed in Chapter 3, the mechanisms necessary for the refinery to function properly can be grouped in three sets of activities: *identification, storage* and *communication* of experiences [226]. Communication activities facilitate the interaction between the refinery and its users. They can be broken down into two sets of tasks: collecting information and dispensing experiences. Through the identification activities, collected information is processed to extract knowledge, for example, to detect lessons learned, and/or develop process models. Storage activities are concerned with packaging and saving models created during identification as well as saving collected information. Hence, an environment supporting the refinery is required to: *collect* information and *package* and *dispense* experiences. These functional requirements can be broken down as follows (see Figure 7.1):

F.1 Collect information: In this set of tasks, information captured from process enactions is recorded in order to document hands-on experiences. Recorded information has to be somehow validated to assure its objectivity; for example, sending questionnaires to as many developers as possible and using statistical techniques to determine trends in the replies.



The Competency Refinery tasks
Collect Information
Package knowledge
Dispense experiences
Capture
Classify
Store
Catalogue
Analyze
Store
Retrieve
Browse
Record
Validate
Characterize
Reason
Match
Select

Figure 7.1: Decomposition of the experience administration tasks

Collected information is then classified according to a predefined taxonomy and stored for future reference. This information should provide evidence of success or failure of a particular process and/or technique.

F.2 **Package knowledge**: Through analysis, the refinery maintainer compiles experiences by exploring knowledge contained in the collected information. Compiled experiences are then catalogued according to the problem they address and features that set them apart from similar experiences. Composed experience packages are then stored in the experience base.

F.3 **Dispense experiences**: The refinery users access the experience base to search for possible solutions for their problems. They should be able to review and reason about the proposed solutions before selecting one. Allowing the users to browse the contents of the experience base is important in case the reasoning system failed to provide a solution to the problem in hand.

7.2.2 System level requirements

In the following we present a list of requirements which we found important for a system to support effectively the refinery's functionality from a technical perspective. The list is compiled from our experience in the enactment of the refinery as well as the requirements reported in similar research results [7] [24] [102] [167] [196]. A proposed system should:

S.1 Support the evolutionary nature of the software experiences. As more experiences are solicited from projects, more of the tacit process knowledge is recognized and explicitly documented. Furthermore, development processes evolve to match the dynamics of the software business and the organizations maturity. This means that packages stored in the experience base should be continuously maintained. For example, at the beginning of our case study, we anticipated the existence of an effective process for developing applications using the framework but we did not have any recommendations to this effect. After the first term, we realized a correlation between the number of developers directly interacting with the framework and the team performance and reflected this knowledge in our recommendations. Towards the end of the case study, it was evident that consultant-based team management is suitable for this type of development. A system to support this form of knowledge evolution must allow the packaging and repackaging of knowledge at different levels of abstraction.

S.2 Support different project setups. The assumption that a standard setup must be used for each project may create barriers to supporting the development process. A competency refinery system must support the development organization as its development practices mature. Hence, the refinery should be able accommodate a variety of project setups by supporting acquisition and deployment techniques that are flexible enough to integrate easily with several development environments. For example, we relied heavily on questionnaires to acquire knowledge about the process performance. For an organization with mature development processes, process measurements might be a better approach to acquire the same information.

S.3 Support evaluating the applicability of packaged knowledge. The dynamic nature of the software business frequently raises the need for new information to be collected and packaged; this, in turn, may render some existing packages obsolete. A package may be obsolete due to the emergence of a better solution for the problem or the problem it addresses no longer exists. For example, the experience base centered around the CSF matured as a result of the information sharing reviews. If this experience base reaches a certain level of maturity, browsing the experience base might be a better approach to support the CSF understanding than reviews. In order to evaluate the best matching knowledge item, the system should support mechanisms to solicit user feedback about the used knowledge items. Maintaining statistics about the system usage (e.g., frequency of search, frequency of access) is also important to differentiate between obsolete knowledge items and items that are not accessed due to problems in the built-in knowledge classification technique.

S.4 Support retrieval of packages of similar experiences. Differences between software development projects do exist, and therefore it is very unlikely to find a packaged experience that exactly matches the current project's characteristics. Typically, software practitioners address a problem by thinking of situations where similar problems occurred and often adapt a previous solution to the current problem. This reuse strategy is commonly used by software developers. For example, based on our experience, it is natural to recommend reviews as a tool for framework understanding despite the product size, the development setup (industrial or academic). In support for this strategy, if the project characteristics can not be exactly matched, the retrieval mechanism should be able to find and rank similar experiences.

S.5 Interactively guide users through the retrieval process. In order to locate the best matching solution, the built-in knowledge classification structure needs to be understood. Often, the refinery users can only anticipate a subset of these classifiers; classifiers such as chances of short-term and long-term success are sometimes used as process classifiers [159], yet they are not easy to anticipate. Knowing and understanding the built-in classification structure should not be required in order to effectively use the system. To overcome these impedances, the system should guide users through the package selection by asking specific questions about the problem context to retrieve the best solution(s) that matches the problem and its environment characteristics.

S.6 **Support retrieval based on incomplete information.** As experience packages stored within the system evolve, more classifiers about these packages emerge. However, the refinery users might not be aware of all the classifiers for the target experience package. The system should be able to retrieve the packages, even if the user did not answer some of the guiding questions.

S.7 **Rely on familiar technologies to acquire and disseminate knowledge.** The steep learning curve needed to use a software tool effectively is a major reason for rejecting the software tool in practice [159] [27]. The refinery system must be easy to learn and use (e.g., by presenting a familiar look and feel on a web platform).

7.3 A prototype environment to support the Competency Refinery

Because a wide range of activities are required in the operation of the refinery, an automated environment should be considered (developed or acquired) to provide basic support for the majority of these activities. First we describe the core technology used to support knowledge deployment, followed by a discussion of the tool architecture we used in our prototype environment. The architecture is generic, scalable and supports not only the dissimilation of experiences, but complete experience reuse as well.

7.3.1 Supporting Technology

As discussed in chapter 2, three candidate reasoning technologies: rule based, model based and case based, can be used to support knowledge deployment activities. In our approach, we favored case-based reasoning for the following reasons:

- Rule-based and model-based reasoning imply that the domain knowledge is well enough understood either to enumerate a causal model or to encode the knowledge into rules; an implication that does not generally hold true for the software process domain.
- Case-based systems can propose solutions without a full understanding of the domain characteristics [136]. They provide an opportunity to make assumptions and predictions about the domain according to what worked in the past.
- Communicating cases, as opposed to rules or model characteristics, back and forth with project participants (domain experts) is relatively simple and straightforward.
- Case-based systems are easier to build and maintain [212].

We also assume that by adding more and more cases to the case base, the domain will become better understood to the point that efficient, comprehensive rule-based systems or models tailored to the organization needs can be developed and enacted for the well-understood aspects of the development processes.

7.3.2 Tool Architecture

A refinery system consists of three layers: data, servers and tools, as shown in Figure 7.2. We distinguish between two types of tools, general purpose tools and experience specific tools. General purpose tools administer the knowledge deployment activities. Specifically, these tools focus on the experience base and allow the refinery customers to locate experiences that meet their needs by

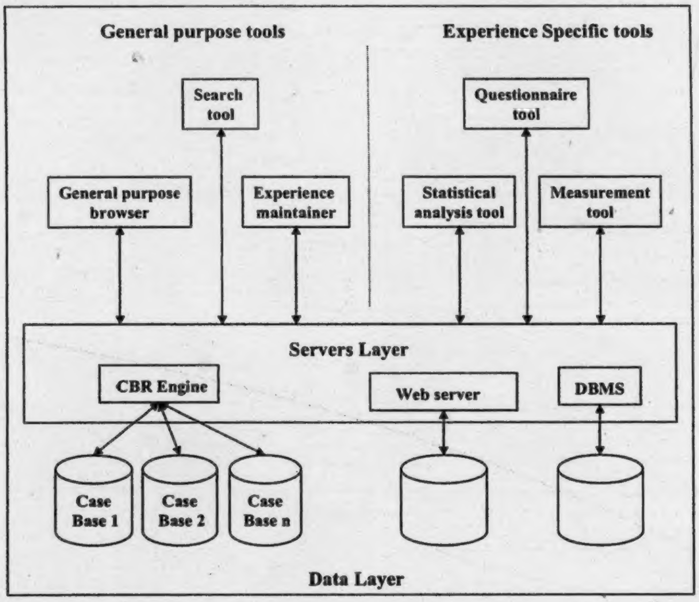

General purpose tools
Experience Specific tools
Search tool
Questionnaire tool
General purpose browser
Experience maintainer
Statistical analysis tool
Measurement tool
Servers Layer
CBR Engine
Web server
DBMS
Case Base 1
Case Base 2
Case Base n
Data Layer

Figure 7.2: Architecture of the competency refinery support environment

searching or browsing the experience base. Other tools in this category include experience input/maintainance tools. These tools support the functionality of different refinery agents. For example, experience input tool support the functionality of the Experience Manager and the search tool supports the functionality of the Experience Adapter. Experience specific tools support the evolution of knowledge contained in the experience base. Tools in this category support the collection of software engineering measurements, data analysis and the polling of feedback from experience users. These tools mainly support the functionality of the Package Developer. For example, the statistical analysis tools developed to analyze the case study data are examples of this category of tools. They focus on analyzing the reviews data to abstract general knowledge about the information sharing reviews.

Both types of tools act as clients using servers within the servers layer. The servers layer contains at its core a case-based reasoning engine to support experience deployment. To manage the amount of knowledge in the process domain, we implemented the experience base as a collection of case bases.

However, in many instances data collected from projects did not fit directly into the experience base (e.g., questionnaire results, review questions and measurements collected about the experience packages). This data is vital for the proper evolution of knowledge within the experience base. It is important to maintain and manage the databases and file systems containing this data within the support environment. Supporting the collection and maintenance of this data requires other types of servers within the servers layer such as Web server, or a database server supported by a Data Base Management System (DBMS).

7.4 System implementation

We built our prototype using a commercially available case-based reasoning system (CASEADVISOR^{TM}) [242]. Several case bases have been built, each targeting a different type of experience. The whole system was then integrated using the Web technology and Pearl scripts to allow the navigation from one experience base to another.

CBR Engine

CASEADVISOR© is an intelligent problem diagnosis and resolution system for applications in enterprise knowledge management. Although the core technology of the system is case-based reasoning engine, it contains additional features such as decision trees and constraint satisfaction algorithms. CASEADVISOR was developed by the CBR group at Simon Fraser University using case-based reasoning technology. Stand-alone and client-server versions of the tool are available to work on either a PC or via an internet connection. In addition to the engine, CASEADVISOR contains two main tools: a CASE AUTHORING tool to build a case base and a PROBLEM RESOLUTION tool to use the case base.

CASE AUTHORING tool

Domain experts interface with the system using the CASE AUTHORING tool. In addition to the case name, the tool provides two generic parts to describe a case: *problem description* and *problem solution*.

Feature-value pairs can be added to increase case distinguishability in the form of questions-answer pairs. Answerers can be individually weighted to index the cases in the PROBLEM RESOLUTION module.

Accessories such as files, decision trees and keywords may be attached to any case to simplify and/or structure the retrieval process or to clarify the proposed solution. Keywords can be manually entered or automatically extracted by the system.

PROBLEM RESOLUTION tool

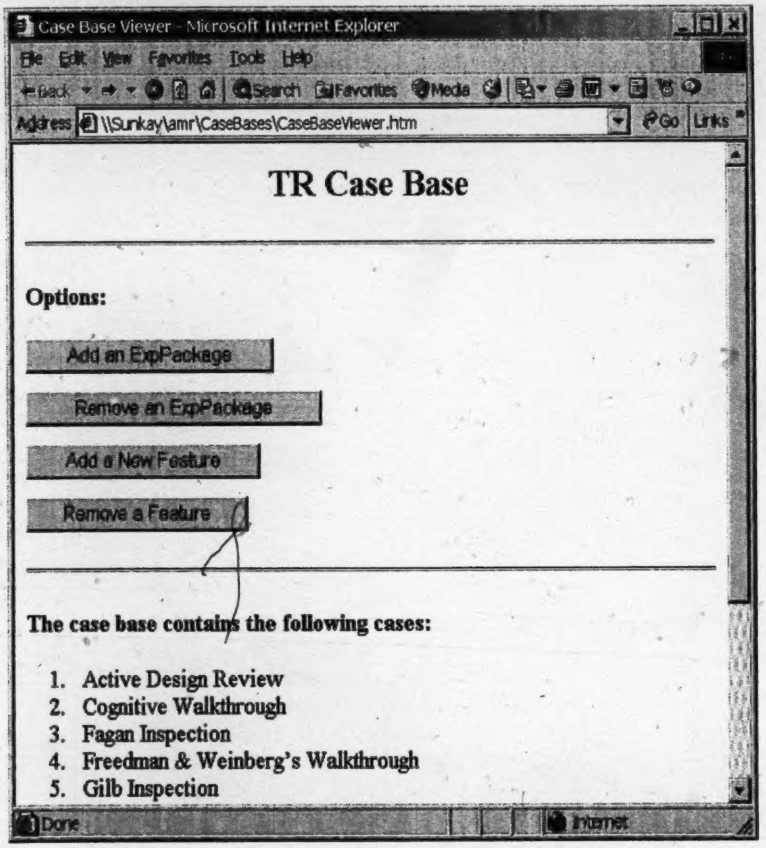
After a case base for a domain has been constructed using the CASE AUTHORING tool, it can be used to solve problems in that domain using the PROBLEM RESOLUTION tool. A user first gives a high-level natural-language description of the problem. The system isolates keywords from the description and uses them to retrieve the cases that best match the description. The questions serve as a logarithmic indexing structure which dynamically re-ranks all retrieved cases. A nearest neighbor formula is used to compute similarity.

Experience maintainer tool

The effect of experience base maintenance activities can be local (affects a particular experience package) or global (affects all packages). Typical maintenance activities for the experience base are [171]:

- Add newly acquired knowledge.
- Remove obsolete packages.
- Add/modify a context parameter or a feature to the experience base.

To facilitate experience maintenance activities we built an experience maintainer tool using perl and CGI scripts. Maintenance processes start with


Case Base Viewer - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address \\Sunkay\amr\CaseBases\CaseBaseViewer.htm
Go Links
Add an ExpPackage
Remove an ExpPackage
Add a New Feature
Remove a Feature
Done
Internet

TR Case Base

Options:

The case base contains the following cases:

1. **Active Design Review**
2. **Cognitive Walkthrough**
3. **Fagan Inspection**
4. **Freedman & Weinberg's Walkthrough**
5. **Gilb Inspection**

Figure 7.3: Maintenance tool screen

extracting experience packages currently stored in the experience base. A summary of the information along with maintenance activities options are presented to the maintainer as in Figure 7.3. The summary lists all package names and, package features along with the values that the feature may take. Depending on the nature of the required update, the maintainer will be presented with forms to either update the information in a single experience package or add a new feature to the experience base. If a new feature is added, the next step is to associate a value for that feature for the experience packages. Finally, the updated information is posted back to the experience base.

Browser tool

The hierarchical nature of the stored experiences requires browsing along two dimensions: (1) within the same case base or (2) from one case base to another. In addition to browsing the same case base, the need for mechanisms to navigate through different levels of experiences was evident.

The browser module was implemented to take full advantage of CASEADVISOR features. Browsing the same case base is provided by default in the PROBLEM RESOLUTION module. For inter case-base browsing we used the "invoking files" feature in CASEADVISOR case description. In the solution description, experience packages are set to invoke and run a new PROBLEM RESOLUTION module using the target case base. At present, browsing is limited to one step either up or down the experience base hierarchy.

Data acquisition tools

Data acquisition represents a major factor that determines the success or failure of a competency refinery. If data acquisition does not happen naturally (i.e. in a transparent manner) as part of the process, developers will refrain from making the additional effort of entering data into the tool. Moreover, the tool has to assert the objectivity of the data solicited from developers.

In our implementation of the refinery, we used a variety of methods to collect data and to assert its objectivity. Two questionnaires were used to solicit feedback from the reviewers about the enacted review process (see Appendix C). The review documentation (e.g., preparation reports and meeting minutes) were collected to analyze the review performance and the review sessions were video taped for further analysis.

Because the knowledge related to CSF is relatively mature, we relied on 'consulting experts' technique while building the experience base for the CSF framework. In this case, a large set of documented problems were solicited from users before they were addressed by the framework expert who provided proposed solutions to these problems. Project members had a chance to interact with the framework expert in a formalized setting (peer review) at the beginning of the project. As a follow-up support process, more problems were addressed through e-mails and face-to-face meetings with the CSF expert.

Data analysis tools

From our experience we soon realized the importance of collecting a set of quantitative and qualitative data through measurements and questionnaires to understand and develop processes that best suit the organization's needs. Collected data were statistically analyzed using Microsoft Excel and a statistical analysis tool called S-Plus. Subjective judgement was necessary in some cases to determine the preference of one process alternative over the others. For example, measurement data in the first enaction of the peer review process in our experiment indicated no difference in performance between the two checklists used; however, post-questionnaire replies indicated that reviewers were more comfortable using the scenario based checklist.

7.4.1 Current status of the experience base

To manage the different levels of knowledge abstraction in our model, we implemented the experience base as a collection of knowledge bases. A knowledge base is dedicated to best practices (praxis package type). Moving down a level of abstraction, a knowledge base is dedicated to knowledge about how to enact a particular process (modus packages). Finally, concrete packages were collected in yet another knowledge base. We found that partitioning the knowledge in this manner helps to produce an experience base with manageable sized knowledge bases. That strategy subsequently eased further analysis of the information. At present, we have encoded three different knowledge bases: *Rapid Development Best Practices (RDBP)* with 27 cases and 3 discriminatory features, *Peer Reviews (PR)* with 18 cases and 4 discriminatory features and *Information Swapping Concrete Experiences (ISCE)* with 15 cases and 7 discriminatory features.

Packages in the (RDBP) (27 packages) were acquired from the set of best practices in rapid development methodology identified by McConnell [159]. These practices are directly associated with development speed and process visibility. To date, all modus packages existing in the experience base exist on the PR knowledge base. In addition to the review package identified in Chapter 6, this knowledge base contains different inspections, technical reviews and walk-though mechanisms proposed by researchers and industry expert (see Appendix B). The 15 packages in the ISCE were acquired from the case study discussed in Chapter 5.

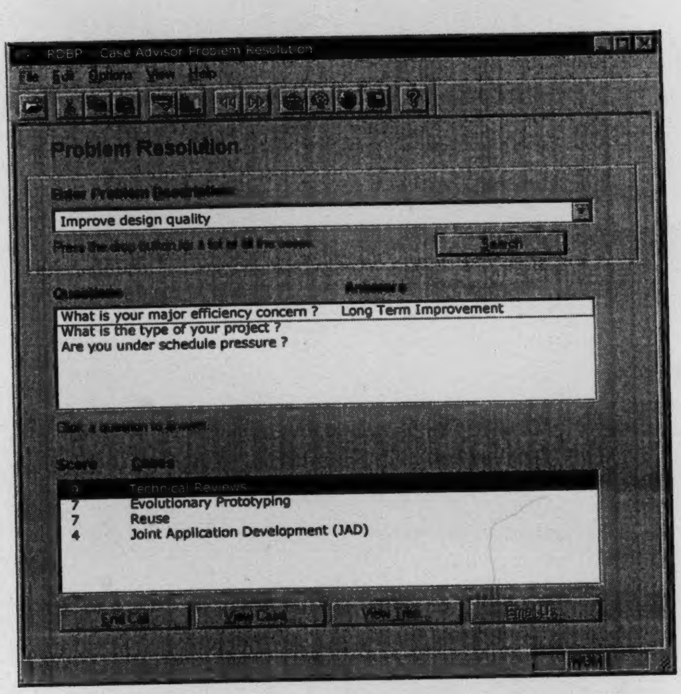
7.5 Usage scenarios

These scenarios demonstrate how the tool is used in practice. The first scenario describes how a typical customer of the refinery would use the support environment to find the most similar experience package. The second scenario describes how the tool is used by the refinery agents to evolve the knowledge contained in the experience base.

7.5.1 Selecting an experience package

Let us assume the project manager of a particular project decides to use some new technique to enhance the design quality. After querying the RDBP experience base, he found that technical reviews is the most appropriate technique to the current project goals (see Figure 7.4). By viewing the details of 'technical reviews' he finds out that there are several technical review processes that could be used. To find out what technical review process best suites the project needs, the project manager invokes the TR experience base and compares different review methodologies to find a set of methodologies that best suit the project context and needs. From this set, let us assume he chooses Fagan Inspection. At this point the project manager may view the Fagan Inspection case (see Figure 7.5) or invoke the concrete Inspection experience base to review hands-on experiences within the organization. By retrieving these experiences, information about the effect on the project schedule, and design quality are retrieved and assessed.

After reviewing all information related to Fagan Inspection, the project manager may either deploy the process or decide that this inspection process is

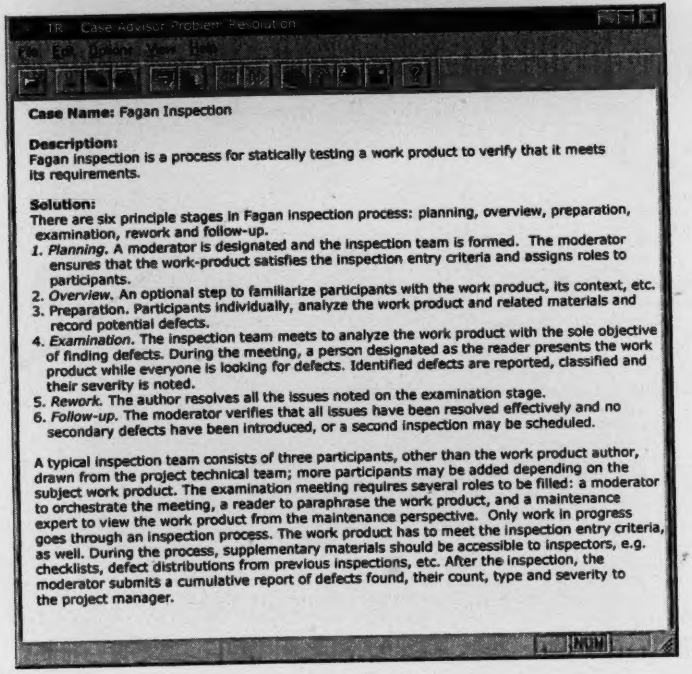


Improve design quality

Questions	Answers
What is your major efficiency concern ?	Long Term Improvement
What is the type of your project ?	
Are you under schedule pressure ?	

Score	Cases
9	Technical Reviews
7	Evolutionary Prototyping
7	Reuse
4	Joint Application Development (JAD)

Figure 7.4: Query the RDBP experience base



Case Name: Fagan Inspection

Description:
Fagan inspection is a process for statically testing a work product to verify that it meets its requirements.

Solution:
There are six principle stages in Fagan inspection process: planning, overview, preparation, examination, rework and follow-up.

1. *Planning*. A moderator is designated and the inspection team is formed. The moderator ensures that the work-product satisfies the inspection entry criteria and assigns roles to participants.
2. *Overview*. An optional step to familiarize participants with the work product, its context, etc.
3. Preparation. Participants individually, analyze the work product and related materials and record potential defects.
4. *Examination*. The inspection team meets to analyze the work product with the sole objective of finding defects. During the meeting, a person designated as the reader presents the work product while everyone is looking for defects. Identified defects are reported, classified and their severity is noted.
5. *Rework*. The author resolves all the issues noted on the examination stage.
6. *Follow-up*. The moderator verifies that all issues have been resolved effectively and no secondary defects have been introduced, or a second inspection may be scheduled.

A typical inspection team consists of three participants, other than the work product author, drawn from the project technical team; more participants may be added depending on the subject work product. The examination meeting requires several roles to be filled: a moderator to orchestrate the meeting, a reader to paraphrase the work product, and a maintenance expert to view the work product from the maintenance perspective. Only work in progress goes through an inspection process. The work product has to meet the inspection entry criteria, as well. During the process, supplementary materials should be accessible to inspectors, e.g. checklists, defect distributions from previous inspections, etc. After the inspection, the moderator submits a cumulative report of defects found, their count, type and severity to the project manager.

Figure 7.5: Case description for Fagan inspection

not appropriate because, for example, it is very human intensive and stretches beyond the project's available resources. At this point, s/he can reinvoke the RDBP experience base and query it again to find out other practices that can positively affect the design quality and adhere to the project's constraints.

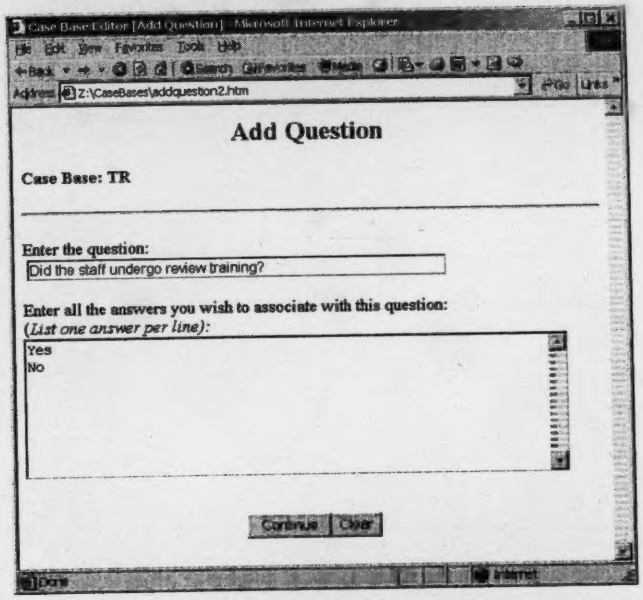
7.5.2 Add a new feature

Assume that analyzing available concrete packages for technical reviews reveals that the chance of successfully enacting certain technical review methodologies highly correlates with formal staff training. To incorporate this new feature "requires formal training" in the TR experience base, we begin by adding the question "Did the staff undergo review training?" and all its plausible answers: "Yes" and "No" (see screen shot in Figure 7.6). After hitting continue, the script will present the domain expert with another form (see screen shot in Figure 7.7) that contains all technical review methodologies stored in the case base with the option of associating the new question with all stored cases and individually setting the weights of each answer. Finally, the question and the weights of the answers for all the associations are posted to the experience base by hitting 'continue' button.

7.6 Assessment and proposed modifications

Initial assessment of the prototype, based on the set of requirements identified earlier, indicate that the major activities of the competency refinery can be supported. Specifically, the prototype supports, to a large extent, most of the identified requirements, however, it fell short of supporting requirement **S.2** (support different project set-ups) and **F.1** (collect information) and provided limited support for requirement **F.2** (Package knowledge) and **S.7** (see table 7.1).

Currently, support for the knowledge deployment tasks is provided through the stand alone version of CASEADVISOR.^{TM} This choice limited our ability to integrate the environment with different project set-ups because in this version of CASEADVISOR,^{TM} the user interface and the CBR engine are tightly


Case Base Editor [Add Question] - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address
Go Links
Continue
Clear
Done
Internet

Z:\CaseBases\addquestion2.htm

Add Question

Case Base: TR

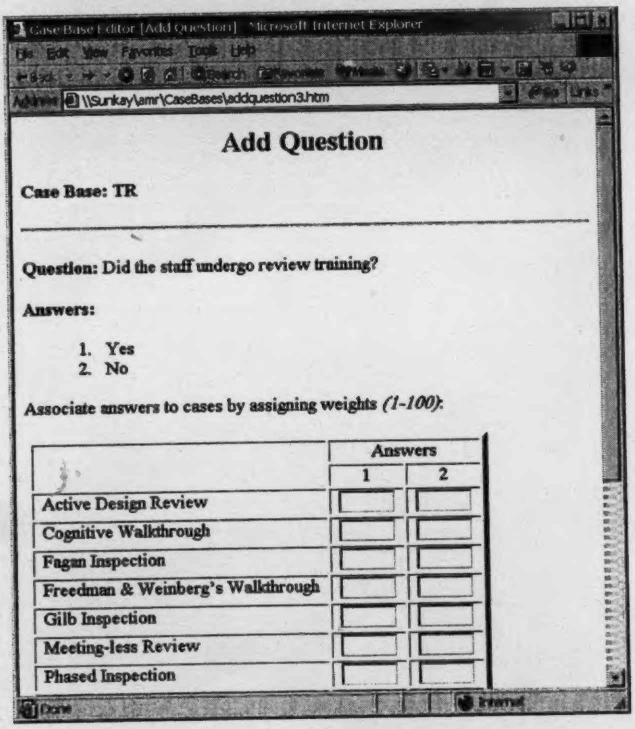
Enter the question:

Did the staff undergo review training?

Enter all the answers you wish to associate with this question:
(*List one answer per line*):

Yes
No

Figure 7.6: Add a new question


Case Base Editor [Add Question] - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address
Go Links
Done
Internet

\\Sunkay\amr\CaseBases\addquestion3.htm

Add Question

Case Base: TR

Question: Did the staff undergo review training?

Answers:

1. **Yes**
2. **No**

Associate answers to cases by assigning weights *(1-100)*:

	Answers	
	1	**2**
Active Design Review		
Cognitive Walkthrough		
Fagan Inspection		
Freedman & Weinberg's Walkthrough		
Gilb Inspection		
Meeting-less Review		
Phased Inspection		

Figure 7.7: Associate questions with cases

Req.	Rating	Comments
F.1	Poor	data collection is not automated
F.2	Limited	developed packaging tools are not general
F.3	Good	
S.1	Good	
S.2	Poor	the prototype works as a stand alone system
S.3	Good	
S.4	Good	
S.5	Good	
S.6	Good	
S.7	Limited	the CBR engine uses an unfamiliar user interface

Table 7.1: Assessment of the prototype

coupled. However, this problem could be solved by using the web-based version of (CASEADVISOR^{TM})[1]- note, when we started developing the prototype the web-based version was not available.

To increase the provided support for incremental learning, better mechanisms are needed to enhance the flow of knowledge to and from the prototype, as well as to improve the contents of the knowledge base. Specifically, we need to:

- foster knowledge acquisition mechanisms that are transparent to the development process.
- integrate the user interface of the knowledge deployment prototype into the development environment supporting the project organization.
- develop a richer knowledge base both at the abstract and concrete levels.

Knowledge acquisition techniques could be the factor causing the success or failure of the refinery support environment. Effective knowledge acquisition techniques have to: (1) be transparent to the development process and, (2) target useful information. Knowledge acquisition techniques that require extra documentation effort are bound to fail as most software professionals view documentation as a cumbersome additional task. Knowledge acquisition tasks have to be transparent to the development process. This could be achieved by using existing project documentation (e.g., project plans, e-mail threads) and process measurements (e.g., costs and benefits) as the main sources of information. Furthermore, by modifying and standardizing parts of these documents (e.g., through document templates) targeted information could be captured.

---

[1] http://www.cs.sfu.ca/~isa/isaresearch.html#systems

For end users to accept and value the knowledge stored in the system, the system has to capture information that is perceived as useful by these users. The system users should be able to report, react to and/or resolve insufficiencies and breakdowns of the stored experiences. Typically, not all users will be interested or willing to influence changes to the experience base; however it is safe to assume that there often exist local users [170] who are interested and capable of performing these tasks.

To increase the value of the experience base, a richer knowledge base has to be developed. At the concrete level, enriching the knowledge base can result from packaging each enaction of the process, as hands-on experience is the source of concrete experience packages. However, including each enaction of the process as a new package should be considered with caution as it may generate inconsistencies in the captured knowledge. For example, performance of individual review sessions in our experiment varied widely. Including these experience packages without performing a root cause analysis to identify the reasons for success or failure may result in inconsistencies in the stored knowledge about the enacted review process.

At the abstract level, the knowledge base can be enriched by encoding more results from work in the software process quality and improvement domain. For example, ISO 15504 (Part 2) [115] (also called SPICE) defines a set of 40 processes that covers most of the software development process. By encoding these processes into praxis level experience packages and seeding them into the knowledge base, a comprehensive list of experiences covering major aspects of the software process could be integrated into the experience base.

For a reasonable-size experience base, end users might not be fully aware of what the refinery has to offer. We found that the existence of a human expert to lead the knowledge deployment tasks is very important to raise users' awareness about the stored experiences. However, the use of human expert is typically expensive and may become the bottle-neck in knowledge deployment tasks. To overcome this anticipated difficulty, we recommend integrating the knowledge deployment tasks into a process support environment such as the cafe-401 environment.[2] The cafe environment is a distributed development environment built using Prothos framework by Avrasoft). This integration will support both knowledge acquisition and knowledge deployment activities.

7.7 Summary

In this chapter we presented an environment to support the Competency Refinery. To support knowledge management and deployment, we implemented a prototype for an automated decision support system using case-based reasoning technology. Assessment of the prototype indicated positive potential for reasoning about the software process as our CBR-based approach can propose solutions without a full understanding of all factors affecting the process. The prototype fell short of satisfying all identified requirements. Specific modifications to the prototype were proposed to fill these gaps.

Apart from the mechanisms to support knowledge acquisition and deployment tasks, we realized the need for productive approaches for knowledge abstraction. These approaches should address the problem that frequently faces the refinery maintainers: Given a set of concrete packages, what abstractions can be made to add or modify a modus package using automated or semi-automated techniques?

---

[2] http://peoria.cs.ualberta.ca:8162/prothos/401_2003_W.x/login.p?

Chapter 8

Conclusions and future work

8.1 Summary

Despite the reported success of the 'accumulation of experiences' paradigm in quality and process improvement programs, successfully deploying the paradigm still represents a major challenge that is undertaken by very few software organizations. This thesis has investigated challenges facing software organizations to establish an environment to manage their experiences and to facilitate collective learning from these experiences; in other words, become a Learning Software Organization (LSO).

The thesis began by critically reviewing existing trends in starting LSO(s) with the purpose of identifying the main challenges in establishing an LSO; two aspects were identified. The first is related to the accumulated experiences. Little information has been delivered about how experiences can be accumulated, explicitly represented, formalized, stored and used in day to day process management [106] [103]. To help address this deficiency, a three level structure was defined for the experience base to capture different classes of knowledge namely: praxis, modus, and concrete package types. Moreover, an experience package template was defined to capture features of accumulated experiences.

The second challenge is related to the actual building of an environment to manage experiences. While the concepts have been treated theoretically, reported efforts indicate that available information is too abstract to effectively implement such an environment [106] [222]. To help address this problem, the thesis defined the competency refinery paradigm to serve as a basis for managing software experiences. To illustrate the competency refinery approach the concept was deployed in a two-year case study to support application development using frameworks, with a special attention given to peer reviews.

Throughout the case study, the refinery was deployed using human agents. The deployment successfully refined the information-sharing review process, supported the development of better documentation for the framework and helped in identifying team management models that provide best value to the development team in the context of the study.

The experience gained from the case study, was then used to define the requirements for a system to automate key aspects of the agents' functionality; namely tasks related to experience storage and dissemination. A prototype environment was then built to fulfill these requirements. Although, the relatively small number of experiences currently stored in the experience base, and the scale and context of the projects used in the case study did not allow a full evaluation of the developed prototype, the prototype was critically assessed. The assessment indicated that most of the important requirements were met; however, a need to improve on acquisition techniques, integrate the environment with a process support system, and develop a richer knowledge base were identified.

8.2 Contributions and results

This dissertation has made several contributions in the area of software quality management and process improvement. In particular, we introduced the Competency Refinery concept to support the re-enactment of successful development activities. The concept was then used to improve peer reviews practices. An experience base for peer reviews was built to provide the required background for the research.

The experience base was used to define an information sharing peer review. The defined process was enacted in 15 different projects, over three consecutive terms, to speed up the learning curve for a moderately complex framework (CSF). By collecting and packaging students' experiences, the applicability of the defined process was examined and validated. Collected experiences were used to suggest process modifications to be enacted in the next round of projects. In addition, the case study collected information to identify successful 'patterns of use' for building application using frameworks, and for improving the quality of the framework documentation. The thesis also introduced an automated environment to support the collection, management and dissemination of packaged experiences.

8.2.1 Building and running a Competency Refinery

The applicability of the concepts underlying the Competency Refinery were examined by providing a concrete implementation for the refinery. By collecting, packaging and managing projects' experiences (15 different projects), the concepts underlying the Competency Refinery were evaluated. By starting with the simplest solution that works, as recommended by agile development methodologies [27], we deployed the refinery using human agents. We then proceeded towards the automation of certain aspects of the agents' tasks.

We found that through the support of a human experience adapter agent, the model could adequately support identification and reenactment of successful development activities. Furthermore, instantiating the refinery using the bottom-up approach proved to be an appropriate decision; we started by collecting concrete data, then abstracted the data to answer specific questions regarding the quality of the peer review process. Following this start-up approach: (*i*) the refinery provided value to its customers as soon as the data was collected; (*ii*) there was no need to define a process ontology, as the development environment was not stable enough to define one.

8.2.2 Packaging process experiences

In order to fulfil different interests of the refinery users, three levels of abstraction were introduced to package process knowledge: praxis, modus and concrete packages. Generally, praxis packages capture the merits of the industry best practices; modus packages represent methodologies of enacting these practices, and concrete packages describe the experience gained by participation on process enactions. The main concept behind this knowledge structure is to simultaneously capture and maintain abstract and concrete process knowledge. The advantages of this structure are:

- Both the knowledge and its roots are captured and stored in the knowledge base.
- Experience from internal and external sources can be fit easily into the structure. Only the internal experiences are supported by concrete knowledge.
- The knowledge base is refined as soon as a concrete experience is captured, rather than waiting until the experience is abstracted and packaged into the experience base.
- Abstracted knowledge can be continuously reevaluated based on the accumulation of concrete experiences, supporting the main concept of the competency refinery paradigm.
- Abstract knowledge is packaged in a variety of ways to fulfil different needs of its users. Abstracted knowledge may be packaged as, modus or praxis type experience depending on intended use.

8.2.3 Automated support for the Competency Refinery

Through our experience in collecting, managing and disseminating knowledge related to the case study, we identified the requirements of an environment to support knowledge management and deployment. A proof of concept environment to support the acquisition, selection, deployment and evolution of experience packages was developed. The environment was implemented as a decision support system using case-based reasoning technology. Although the environment did not to provide general approaches for abstracting process knowledge, several statistical analysis packages were developed to analyze and abstract knowledge accumulated through the enaction of peer reviews. These packages were developed in S, a language developed at AT&T for statistical analysis. They also can be executed in S-Plus, a commercially available statistical analysis tool or R, an open source statistical analysis environment.

8.2.4 Process taxonomy model for peer reviews

The thesis defined a process taxonomy to distinguish several software processes. The goal of the taxonomy was to meet the needs of the engineering decision making process introduced in Chapter 4. The taxonomy organized the process along three dimensions: technical, economic and support as described in Section 4.4. The taxonomy was applied on peer review processes, and successfully identified 16 different proposed and practiced review processes which were organized in an experience base for peer reviews.

The taxonomy presents an up-to-date overview and analysis of peer review knowledge presented in literature and identified 17 different process attributes for peer reviews as described in Section 4.5. These attributes characterized the core concepts of the review process and allowed the identification of the details of the review process that best suite a particular situation through the comparison of different alternatives along various dimensions.

8.2.5 Documentation of a major case study

A long-term foundational case study was initiated in this thesis. The two-year study investigated processes supporting the infusion of new technologies in a software development organization. Specifically, we examined how software developers, with little or no knowledge of a framework, approached the development of new applications using this framework. At the macro level, the study was used to evaluate the competency refinery concept. At the micro level, the study focused on examining processes and techniques supporting software development using application frameworks.

In addition to confirming the value of process improvement based on the accumulation of experiences, the study showed that: (*i*) peer reviews are useful technique to speed-up frameworks learning; (*ii*) the consultant-based team management model provided the best results for building of applications using this framework; and (*iii*) in addition to the functionality provided by the framework, the users are interested in learning more about the non-functional aspects of the framework (e.g., reliability).

8.2.6 Supporting framework knowledge internalization

Over the course of the case study, the adopted techniques (peer reviews and the framework's experience base) successfully disseminated knowledge about the framework to interested parties. Analysis of quantitative data from our case study showed that reviews are useful interaction mechanism to facilitate framework knowledge internalization at early stages of the application development. These findings were further confirmed by the subjective assessment of the study participants. Reviewers pointed out the benefit of reviews in three areas: (*i*) setting deadlines for the understanding process, (*ii*) consolidating the development team point of view through well-organized discussion, and (*iii*) getting fast feedback from the framework expert. The experience base, centered around the framework, significantly improved the framework documentation by including the users' perspective. Over time, the experience base drastically reduce the users' dependency on the framework expert.

8.3 Future directions

While pursuing this thesis research, a number of interesting questions were uncovered that could extend the research.

8.3.1 Tool development and refinement

The developed prototype shows promise and leads naturally into the possibility of developing a production-quality knowledge management environment. This environment can be the basis for further studies and refinements in the refinery model as well as mechanisms supporting experience base management. Some of the key issues with the environment development that need immediate attention are:

Integration with other tools Knowledge management tools deliver their best performance when integrated into existing development environment. This integration facilitates knowledge acquisition tasks and enhances opportunities for knowledge dissemination. Typically, automated development environments collect a lot of information throughout the development activities. By analyzing and/or modifying the collected information, the refinery knowledge acquisition can happen effortlessly. For example, integrating the prototype described in Chapter 7 with an automated peer review tool will replace many of the specialized data collection forms we developed. Review findings, time consumption and checklist outcomes can all be electronically captured through an integrated peer review tool.

Maintaining the experience base The refinery concept is based on the accumulation of experiences supporting the continuous growth of the experience base and thus emphasizing the importance of deploying tasks to maintain that experience base. The complexity of these maintenance tasks can be attributed to: *i*) the evolutionary nature of the stored knowledge which can quickly render some packages obsolete; *ii*) the diverse sources of cases which can easily generate inconsistencies among stored cases; and *iii*) the emergence of new development methodologies that may require substantial restructure of the experience base. The complexity of these tasks, coupled with the demanding time and performance requirements of the software industry, strongly suggests the need to find better automated support for experience-base maintenance.

8.3.2 Extending the experience base

From his experience in building knowledge units, Schneider concluded that [200] seeding the experience base is a fundamental requirement to start a knowledge unit. Based on our experience, seeding an experience base is not a trivial task. Background research about the target process is required to apply the process taxonomy framework and to reflect the up-to-date overview and analysis of that process.

To exploit the full advantage of the Competency Refinery, the developed experience bases have to cover as many processes as possible. As recommended in Chapter 7 an investigation of how to extend the experience base by aligning it with the SPICE embedded model should be undertaken. Specifically, processes defined in the SPICE model can be used as basis for seeding the refinery's experience base. The SPICE standard defines a sophisticated model of software process management consisting of 40 different processes drawn from the the world-wide experience of large and small software organizations.

8.3.3 Case studies

The case study administered as part of this thesis enriched the peer-reviews experience base by giving insights about how reviews can support frameworks understanding. Despite the number of projects included in the study, the scope of the study was somehow limited. It focused on peer reviews, was run using one framework, and took place in an academic context. More case studies using different frameworks, larger projects, different development context and/or focusing on different activities supporting framework understanding (e.g., structured tutoring) need to be explored.

8.3.4 Controlled experiments

Our initial assessment of the tool indicates positive results supporting the refinery's functionality. However, more controlled experiments are required to determine the areas of strength and weaknesses of the tool as it compares with deployment through a human agent.

8.3.5 Documenting application frameworks

Frameworks are bound to evolve as they mature and more is understood about the domain they represent. Nowadays, framework evolution is an active research areas in software engineering. Framework documentation needs to evolve to reflect changes in the framework, as should the experience base built around the framework. More work is needed to determine when and how the experience base will be affected by the evolution of the underlying framework. For example, a model (formal or informal) that ties cases in the experience base with changes in the associated framework could be used to determine hot spots in the framework and predict framework evolution.

8.3.6 Peer reviews for framework understanding

Although information swapping and learning is a well recognized benefit of reviews [110], little has been done to explore the full capacity of reviews to serve these objectives. Most of the research concerning peer reviews has focused on defect detection capabilities. More research is needed on reviews that have information swapping and learning as a central objective. For example, research is needed to answer the following questions:

- **How to form an effective checklist?** Our results indicated no difference between the two checklists used; however, many researchers [92], [53] emphasized the value of the checklist in supporting reviews. More studies are also required to track and evaluate the effect of checklist improvement techniques (e.g., the statistical method described in [53]).
- **What data to collect?** Gathering data concerning the performance of the review process is essential for its improvement. Gilb & Graham [92] define over fifty measures to collect. [1] In order to define a comprehensive set of measures that will reflect the actual performance of information sharing reviews, research is needed to evaluate the usefulness and coverage of the measures defined by Gilb & Graham for information sharing and learning purposes.

8.3.7 Forecasting and estimation

Despite the effectiveness of peer reviews in information sharing, reviewers are unlikely to discover all questions they want to ask about a framework. Hence, it would be useful to estimate how many problems (questions) are yet to be discovered. From the application development perspective, the estimate could be used to judge how well the developers understand the framework prior to engaging in the development. From the framework documentation perspective, the estimate could be used to judge the completeness of the documentation. Capture-Recapture techniques [205] [239] have been suggested to estimate number of defects remaining in a document after inspection [73] [163]. However, the robustness of the CR models with respect to reviews need to be carefully examined, as reported research indicated disagreement in the results (see for example [163] and [41]).

---

[1] Other list of measurements defined for reviews could be viewed as subset of this list (e.g., [72], [219]).

8.4 Concluding remarks

From the research performed, it is clear that supporting software quality and process improvement based on the accumulation of experiences is valuable. Experiences accumulated about the difficulties facing the CSF users resulted in framework documentation that reflects the users' perspectives. Monitoring the performance of the information sharing peer reviews resulted in the refinement of a review process that positively affects application development using frameworks. Tracking the overall performance of the projects identified consultant-base team management as an effective model for developing this type of applications. The observed success of the Competency Refinery concepts defined and deployed in this thesis research suggests that these concepts can be used as a baseline for exploring more advanced techniques to support software experience acquisition and management.

The Nunamaker *et al.* [173] methodology for design science research was followed in this thesis and it proved to be suitable for this line of research. Two cycles of this methodology were performed in this thesis. In the first cycle, the Competency Refinery model and its underlying paradigms were defined in the concept building phase. During system development, a refinery for peer reviews was built. The first cycle ended after the peer review refinery was used in a case study to validate the defined concepts. From our experience in this cycle we learned that the interpretation of collected data is not a straightforward process. The data reduction process is time consuming and in few cases we had to use our subjective assessment. We also learned that experiments assessing knowledge deployment is hard to control, as knowledge disseminates along different routes that are not necessarily under the researchers' control.

The second cycle was started by defining the requirements for an automated environment to support the refinery, followed by the development of a prototype environment to satisfy these requirements, and ended by a critical evaluation of the developed environment. While implementing the tools, we learned that having access to the internal representation of data within the CBR system is useful. We also learned that even if the refinery is using an off-the-shelf CBR engine, understanding the underlying indexing mechanism is essential for the proper implementation of the knowledge base.

The cycle of conceptualize/build/evaluate defined by Nunamaker *et al.* can be repeated as required. The only caveats are the size of the developed experience base and the evaluation environment context. Unquestionably, a large experience base is required to explore the full potential of an automated environment supporting the competency refinery. Furthermore, due to time and budget limitations, there is an upper limit to the value achieved from observing student projects. The research cycle would be more effective if continued with software professionals in an industrial setup.

The area of software experience management is still in its infancy; there is much scope for further research. Development of large experience bases, advanced search algorithms, and knowledge abstraction techniques are a few of the topics that need innovative research to boost the already realized benefits of software experience management. As one of the most useful software verification and validation tools, peer reviews is another research area that deserves much more investigation to exploit its full potential. In particular, a critical investigation of the parameters that affect review performance is likely the best means for defining review techniques that can exhibit better/consistent performance.

Bibliography

[1] Aamodt and E. Plaza. Case based reasoning: Foundational issues, methodological variations, and system approaches. *AI Communications*, 7(1):39–59, 1994.

[2] T. Abdel-Hamid and S.E. Madnick. *Software Project Dynamics: An Integrated Approach.* Prentice Hall, New York, NY., 1991.

[3] A. F. Ackerman, L. S. Buchwald, and F. H. Lewsky. Software inspections: An effective verification process. *IEEE Software*, 6(3):31–36, 1989.

[4] A.F. Ackerman, P.J. Fowler, and R.G. Ebenau. Software inspections and the industrial production of software. In H.L Hausen, editor, *Software Validation*, pages 13–40. Elsevier, Amsterdam, 1984.

[5] K-D Althoff, A. Birk, S. Hartkopf, W. Müller, D. Surmann, and C. Tautz. Managing software engineering experience for comprehensive reuse. In *Proceedings of the 11th International Conference on Software Engineering and Knowledge Engineering (SEKE99)*, pages 10–19, 1999.

[6] K-D. Althoff, A. Birk, C.G. von Wangenheim, and C. Tautz. CBR for experimental software engineering. In *Case Based Reasoning Technology - From Foundations to Application*, chapter 9, pages 235–254. Springer-Verlag, Berlin, Heidelberg, 1998.

[7] K-D Althoff, F. Bomarius, and C. Tautz. Using case-based reasoning technology to build learning software organizations. In *Proceedings of the Interdisciplinary Workshop on Building, Maintaining and Using Organizational Memory (OM-98)*, Brighton, UK., 1998.

[8] K-D Althoff, B. Decker, S. Hartkopf, A. Jedlitschka, M. Nick, and J. Rech. Experience management: The Fraunhofer IESE Experience Factory. Technical Report 035.01/E, Fraunhofer Institute for Experimental Software Engineering, Kaiserslautern, Germany, 2001.

[9] K.-D. Althoff, R.L. Feldmann, and W. Müller, editors. *proceedings of the 3rd International Workshop on Learning Software Organization LSO'01*, LNCS 2176. Springer-Verlag, Berlin, Heidelberg, 2001.

[10] K-D Althoff, S. Hartkopf, and W. Müller, editors. *Proceedings of the 2nd International Workshop on Learning Software Organization LSO'00.* Fraunhofer Institute for Experimental Software Engineering, Kaiserslautern, Germany, 2000.

[11] K-D Althoff, M. Nick, and C. Tautz. CBR-PER: A tool for implementing reuse concepts of the experience factory for cbr systems. In *Proceedings of the 7th German Conference on Knowledge Based Systems (XPS99) Workshop on Case-Based Reasoning*, 1999.

[12] D.M. Amidon. *Innovation Strategy for the Knowledge Economy: The Ken Awakening.* Butterworth-Heinemann, Boston, 1997.

[13] C. Anderson. World gone soft: A survey of the software industry. *IEEE Engineering Management Review*, 24(4):21–36, 1996.

[14] M. P. Ardissone, M. Spolverini, and M. Valentini. Statistical decision support method for in-process inspections. In *Proceedings of the 4th International Conference on Achieving Quality In Software*, pages 135–143, 1998.

[15] L.J. Arthur. *Improving Software Quality: An Insider's Guide to TQM.* John Wiley & Sons Inc., New York, NY, 1993.

[16] J. Barnard and A. Price. Managing code inspection information. *IEEE Software*, 11(2):59–69, 1994.

[17] K. Bartlmae. An experience factory approach for data mining. In *Proceedings of the 2nd Workshop in Data Mining and Data Warehousing as Basis of Modern Decision Support Systems*, 1999.

[18] V. Basili, S. Green, O. Laitenberger, F. Lanubile, F. Shull, S. Sorumgard, and M. Zelkowitz. The empirical investigation of perspective-based reading. *Journal of Empirical Software Engineering*, 2(1):133–164, 1996.

[19] V. Basili, M. Lindvall, and P. Costa. Implementing the experience factory concepts as a set of experience bases. In *Proceedings of the 13th International Conference on Software Engineering and Knowledge Engineering (SEKE01)*, pages 102–109. Knowledge Systems Institute, 2001.

[20] V. R. Basili. Evolving and packaging reading technologies. *Journal of Systems and Software*, 38(1), 1997.

[21] V.R. Basili. The experience factory and its relationship to other improvement paradigms. In I. Sommerville and M. Paul, editors, *Proceedings of the 4th European Software Engineering Conference ESEC*, LNCS 717, Germany, 1993. Springer-Verlag, Berlin, Heidelberg.

[22] V.R. Basili and G. Caldiera. Methodological and architectural issues in the experience factory. In *Proceedings of the 16th Annual Software Engineering Workshop, NASA/GSF*, Software Engineering Laboratory Series, Greenbelt, Maryland, 1991.

[23] V.R. Basili, G. Caldiera, F. McGarry, R. Pajerski, G. Page, and S. Waligora. The software engineering laboratory – an operational software experience factory. In *Proceedings of the 14th International Conference on Software Engineering (ICSE14)*, pages 370–378, Melbourne, Australia, 1992.

[24] V.R. Basili, G. Caldiera, and H.D. Rombach. The experience factory. In J.J. Marciniak, editor, *Encyclopedia of Software Engineering.*, pages 468–476. John Wiley & Sons Inc., New York, NY, 1994.

[25] V.R. Basili and F. McGarry. The experience factory: How to build and run one. Tutorial at the 19th International Conference on Software Engineering., 1997.

[26] V.R. Basili and H.D. Rombach. Support for comprehensive reuse. *IEEE Software Engineering Journal*, 6(5):303–316, 1991.

[27] K. Beck. Extreme programming: A humanistic discipline of software development. In *Proceedings of the 1st International Conference of Fundamental Approaches to Software Engineering (FASE98)*, pages 1–6, 1998.

[28] B. Bell, W. Citrin, C. Lewis, J. Rieman, R. Weaver, N. Wilde, and B. Zorn. Using the programming walkthrough to aid in programming language design. *Software Practice and Experience*, 24(1):1–25, 1994.

[29] F. Belli and R. Crisan. Towards automation of checklist-based code-reviews. In *Proceedings of the 8th International Symposium on Software Reliability Engineering*, 1996.

[30] G. Bhatt. Managing knowledge through people. *Journal of Knowledge and Process Management*, 5(3):165–171, 1998.

[31] F. Biemans. Reference model of production control system. In *Proceedings of IECOMN 86*, Milwaukee, Min., 1986.

[32] A. Birk, D. Surmann, and K.-D. Althoff. Knowledge acquisition in experimental software engineering. In *Proceedings of the 11th European Workshop on Knowledge Acquisition, Modelling and Management*, pages 67–84, Dagstuhl Castle, Germany, 1999. Springer-Verlag, Berlin, Heidelberg.

[33] A. Birk and C. Tautz. Knowledge management of software engineering lessons learned. In *Proceedings of the 10th International Conference on Software Engineering and Knowledge Engineering (SEKE99)*, 1998.

[34] D.B. Bisant and J.R. Lyle. A two-person inspection method to improve programming productivity. *IEEE Transactions on Software Engineering*, 15(10):1294–1304, 1989.

[35] B. Boehm. *Software Engineering Economics*. Prentice Hall, New York, NY., 1981.

[36] F. Bomarius, editor. *Proceedings of the 1st International Workshop on Learning Software Organization LSO'99*. Fraunhofer Institute for Experimental Software Engineering, Kaiserslautern, Germany, 1999.

[37] G. Booch. *Object Solutions: Managing the Object-Oriented Project*. Addison Wesley Publishing Company, Reading, MA., 1995.

[38] S. Boodoo, K. El-Emam, O. Laintenberger, and N. Madhavji. The optimal team size for UML design inspections. Technical Report NRC/ERB-1081, National Research Council, Institute for Information Technology, 2000.

[39] J. Bosch. Software product lines: Organizational alternatives. In *Proceedings of the 23th International Conference on Software Engineering (ICSE23)*, pages 91–100, Toronto, Canada, 2001. ACM Press.

[40] K. V. Bourgeois. Process insights from a large-scale software inspections data analysis. *Cross Talk, The Journal of Defense Software Engineering*, pages 17–23, 1996.

[41] Briand, K. L. El Emam, B. Freimut, and O. Laitenberger. A comprehensive evaluation of capture-recapture models for estimating software defect content. *IEEE Transactions on Software Engineering*, 26(6), 2000.

[42] R.N. Britcher. Using inspection to investigate program correctness. *IEEE Computer*, 21(11):38–44, 1988.

[43] F.P. Jr. Brooks. *The Mythical Man-Month.* Addison Wesley Publishing Company, Reading, MA., 1978.

[44] P. Brössler. Knowledge management at a software house: A progress report. In F. Bomarius, editor, *Proceedings of the 1st International Workshop on Learning Software Organization LSO'99*, pages 77–83. Fraunhofer Institute for Experimental Software Engineering, Kaiserslautern, Germany, June 1999.

[45] B. Brykczynski. The software inspection process - Applying the principles of Deming and Crisby. *Information and Systems Engineering*, 1(1):23–37, 1995.

[46] R.D. Buck and J.H. Dobbin. *Software Validation*, chapter Application of Software Inspection Methodology in Design and Code, pages 41–56. Elsevier, Amsterdam, 1984.

[47] T. Burns and G.M Stalker. *The Management of Innovation.* Oxford University Press, London, revised edition edition, 1994.

[48] M. Bush. Improving software quality:The use of formal inspections at the jet propulsion laboratory. In *Proceedings of the 12th International Conference on Software Engineering (ICSE12)*, pages 196–199, Nice, France, 1990. ACM Press.

[49] D. Butler and P. Denommee. Documenting frameworks. In Fayad, Schmidt, and Johnson, editors, *Building Application Frameworks.*, pages 495–503. Wiley Computer Publishing, New Your, NY, 1999.

[50] K. Cai. On estimating the number of defects remaining in software. *Journal of Systems and Software*, 40:93–114, 1998.

[51] NASA Goddard Space Flight Center. Software engineering laboratory database oranization and user's guide. Technical Report SEL-89-101, NASA/GSFC, Greenbelt, Maryland, Feb 1990. Revesion 1.

[52] B. Chatters. Implementing an experience factory: Maintenance and evolution of the software and systems development process. In *Proceedings of the International Conference on Software Maintenance (ICSM)*. IEEE Computer Society Press, Los Alamitos, Ca., 1999.

[53] Y. Chernak. A statistical approach to the inspection checklist formal synthesis and improvement. *IEEE Transactions on Software Engineering*, 22(12):866–874, 1996.

[54] J.A. Clapp, S.F. Stanten, W.W. Peng, D.R. Wallace, D.A. Cerino, and Jr. R.J. Dziegiel. Software quality control, error analysis and testing. Noyes Data Corporation, Mill Roald, Park Ridge, NJ, 1995.

[55] T. Collins. Bank error hands out 2bn pounds in half an hour. *Computer Weekly (UK)*, October 19 1989.

[56] J. S. Collofello and S. N. Woodfield. Evaluating the effectiveness of reliability-assurance techniques. *Journal of Systems and Software*, 9:191–195, 1989.

[57] J.S. Collofello. The software technical review process. SEI Curriculum Module SEI-CM-3-1.5, Software Engineering Institute, 1988.

[58] R. Conradi, C. Fernström, and A. Fuggetta. A conceptual framework for evolving software processes. *Software Engineering Notes*, 18(4):26–45, Oct 1993.

[59] R. Conradi, A.S. Marjara, and B. Skatevik. An empirical study of inspection and test data at Ericsson. In *Proceedings of the International Conference on Product-Focused Software Process Improvement (PROFES'99)*, VTT Symposium : 195, pages 263–284, Oulo, Finland, 1999. VTT.

[60] M.E. Conway. How do committees invent? *Datamation*, 14(4):28–31, 1968.

[61] S.D. Cook and J.S. Brown. Bridging epistemologies: the generative dance between organization knowledge and organization knowing. *Organization Science*, 10(4):381–400, 1999.

[62] P. D'Astous and P. N. Robillard. Characterizing implicit information during peer review meetings. In *Proceedings of the 22th International Conference on Software Engineering (ICSE22)*, 2000.

[63] T.H. Davenport and G. Probst, editors. *Knowledge Management Case Book: Siemens Best Practices*. MCD, Munich, Germany, 2001.

[64] T.H. Davenport and L. Prusak. *Working Knowledge: How organizations Manage What They Know*. Harvard Business School, Boston, 1998.

[65] W. E. Deming. *Out of the Crisis*. MIT Press, Cambridge, MA, 1986.

[66] M. Diehl and W. Stroebe. Productivity loss in brainstorming groups: Toward the solution of a riddle. *Journal of Personality and Social Psychology*, 53(3):497–509, 1987.

[67] E. P. Doolan. Experience with Fagan's inspection method. *Software Practice and Experience*, 22(3):173–182, 1992.

[68] A. Dorling. SPICE: Software Process Improvement and Capability dEtermination. *Software Quality Journal*, 2:209–224, 1993.

[69] H.E. Dow and J.S. Murphy. Detailed product knowledge is not required for a successful formal software inspection. In *Proceedings of the seventh Software Engineering Process Group Meeting*, Boston, MA., 1994.

[70] M. Dyer. Verification-based inspection. In *Proceedings of the 25th Annual Hawaii International Conference on System Sciences*, volume 2, pages 418–427, 1991.

[71] M. Dyer. *The Cleanroom Approach to Quality Software Development.* John Wiley & Sons Inc., New York, NY, 1992.

[72] R. G. Ebenau. Predictive quality control with software inspections. *CrossTalk*, 7(6):916, 1994.

[73] S. G. Eick, M. D. Loader, C. R.and Long, L. G. Votta, and S. Vander Wiel. Estimating software fault content before coding. In *Proceedings of the 14th International Conference on Software Engineering (ICSE14)*, pages 59–65, 1992.

[74] K. El Emam and O. Laitenberger. Evaluating capture-recapture models with two inspectors. *IEEE Transactions on Software Engineering*, 27(9):851–864, 2001.

[75] H. Eriksson. A survey of knowledge acquisition techniques and tolls and their relationship to software engineering. *Journal of Systems and Software*, pages 97–107, 1992.

[76] Experience base. A booklet from the PERFECT ESPRIT project 9090 Handbook Edition, 1996.

[77] M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976.

[78] M. E. Fagan. Advances in software inspections. *IEEE Transactions on Software Engineering*, 12(7):744–751, 1986.

[79] A.V. Feigenbaum. *Total Quality Management.* McGraw Hill Inc., New York, NY, Fortieth Anniversary edition, 1991.

[80] R.L. Feldmann and K-D. Althoff. On the status of learning software organizations in the year 2001. In R.L. Feldmann and K-D. Althoff, editors, *proceedings of the 3rd International Workshop on Learning Software Organization LSO'01*, Lecture Notes in Computer Science 2176, pages 2–6. Springer-Verlag, Berlin, Heidelberg, 2001.

[81] G. Fischer. Seeding, evolutionary growth and reseeding: Construction, capturing and evolving knowledge in domain oriented design environments. *Automated Software Engineering*, 5(4):447–464, 1998.

[82] P. J. Fowler. In-process inspections of work-products at AT&T. *AT&T Technical Journal*, 65(2):102–112, 1986.

[83] L. A. Franz and J. C Shih. Estimating the value of inspections and early testing for software projects. *Hewlett-Packard Journal*, CS-TR-6, 1994.

[84] D. P. Freedman and G. M. Weinberg. *Handbook of Walkthroughs, Inspections, and Technical Reviews.* Dorset House Publishing, New York, 4th edition, 1990.

[85] G. Froehlich. Client-Server Framework.
http://www.cs.ualberta.ca/~garry/framework, 1999.

[86] G. Froehlich. *Hooks: AnHooks: An Aid to the Use of Object-Oriented Frameworks Aid to the Use of Object-Oriented Frameworks.* PhD thesis, University of Alberta, 2002.

[87] G. Froehlich, H.J. Hoover, L. Liu, and P. Sorenson. Hooking into object-oriented application frameworks. In *Proceedings of the 19th International Conference on Software Engineering (ICSE19)*, pages 491–501, Boston MA., 1997. ACM Press.

[88] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley Publishing Company, Reading, MA., 1995.

[89] D. Gangopadhyay and S. Mitra. Understanding frameworks by exploration of exemplars. In *Proceedings of the 7th International Workshop on Computer Aided Software Engineering (CASE-95)*, pages 90–99, Toronto, Canada, 1995.

[90] A.A. Gately. Design and code inspection metrics. In *International Conference on Software Management and Applications of Software Measurement*, San Jose, Ca., 1999.

[91] T. Gilb. *Principles of Software Engineering Management.* Addison Wesley Publishing Company, Reading, MA., 1988.

[92] T. Gilb and D. Graham. *Software Inspection.* Addison Wesley Publishing Company, Reading, MA., 1993.

[93] R.L. Glass. *Building Quality Software.* Prentice Hall, New York, NY., 1992.

[94] M. Goodman. Cbr in battle planning. In *Proceedings of Workshop of Case Based Reasoning (DARPA).* Morgan Kaufmann, 1989.

[95] R. Grady. *Practical software metrics for project management and process improvement.* Prentice Hall, New York, NY., 1992.

[96] R. B. Grady and T. van Slack. Key lessons in achieving widespread inspection use. *IEEE Software*, 11(4):46–57, 1994.

[97] M.L. Griss, J. Favaro, and P. Walton. Managerial and organizational issues - starting and running a software reuse program. In W. Schäfer, R. Prieto-Diaz, and M. Matsumoto, editors, *Software Reusability*, chapter 3, pages 51–78. Ellis Horwood Ltd., 1994.

[98] M. T. Hansen, N. Nohria, and T. Tierney. What's your strategy for managing knowledge? *Harvard Business Review*, pages 106–116, March-April 1999.

[99] M.D. Hansen. Survey of available software-safety analysis techniques. In *Proceedings of the Annual RAM Symposium*, pages 46–49. IEEE, 1989.

[100] J.J Hart. The effectiveness of design and code walkthroughs. In *Proceedings of the International Computer Software and Applications Conference, COMPSAC'82*, pages 512–522, Silver Spring, MD, Nov. 1982. IEEE Computer Society Press, Los Alamitos, Ca.

[101] J. Hartmanis. Turing award lecture: On computational complexity and the nature of computer science. *Communications of the ACM*, 37:37–43, 1994.

[102] S. Henninger. Capturing and formalizing best practices in a software development organization. In *Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering (SEKE97)*, Spain, 1997.

[103] S. Henninger. Using software process to support learning software organizations. In *Proceedings of the 25th Annual Software Engineering Workshop, NASA/GSF*, Software Engineering Laboratory Series, Greenbelt, Maryland, 2000.

[104] W. C. Hetzel. *An Experimental Analysis of Program Verification Methods.* PhD thesis, University of North Carolina at Chapel Hill. Department of Computer Science., 1976.

[105] F. Houdek and C. Bunse. Transferring experience: A practical approach and its application on software inspection. In F. Bomarius, editor, *Proceedings of the 1st International Workshop on Learning Software Organization LSO'99*, pages 59–68. Fraunhofer Institute for Experimental Software Engineering, Kaiserslautern, Germany, June 1999.

[106] F. Houdek and H. Kempter. Quality patterns - An approach to packaging software engineering experience. *Software Engineering Notes*, 22(3):81–88, 1997.

[107] F. Houdek, K. Schneider, and E. Wieser. Establishing experience factories at Daimler-Benz. An experience report. In *Proceedings of the 20th International Conference on Software Engineering (ICSE20)*, pages 443–447, Kyoto, Japan, 1998. ACM Press.

[108] W. S. Humphrey. *A Discipline for Software Engineering.* Addison Wesley Publishing Company, Reading, MA., 1995.

[109] W. S. Humphrey. *Managing Technical People.* Addison Wesley Publishing Company, Reading, MA., 1997.

[110] W.S. Humphrey. *Managing the Software Process.* Addison Wesley Publishing Company, Reading, MA., 1990.

[111] W.S. Humphrey. *A personal Commitment to Software Quality.* Ed Yourdon's American Programmer, 1994.

[112] IEEE standard for software reviews and audits. ANSI/IEEE 1028-1988, 1988.

[113] IEEE standard glossary of software engineering terminology. IEEE 610.12-1990, 1990.

[114] Information technology – software life cycle processes. International Organization for Standardization (ISO), ISO 12207, 1995.

[115] ISO15504. *A reference model for processes and process capability.* International Organization for Standardization, 1997.

[116] C. Johansson, P. Hall, and M. Coquard. Talk to Paula and Peter - They are experienced. In F. Bomarius, editor, *Proceedings of the 1st International Workshop on Learning Software Organization LSO'99*, pages 69–76. Fraunhofer Institute for Experimental Software Engineering, Kaiserslautern, Germany, June 1999.

[117] Ph.M. Johnson. An instrumented approach to improving software quality through formal technical review. In *Proceedings of the 16th International Conference on Software Engineering (ICSE16)*, pages 113–122. ACM Press, 1994.

[118] P.M. Johnson. Reengineering inspection. *Communications of the ACM*, 41(2):49–52, 1998.

[119] P.M. Johnson and D. Tjahjono. Improving software quality through computer supported collaborative review. In *Proceedings of the 3rd European Conference on Computer Supported Cooperative Work*, pages 61–76, 1993.

[120] P.M. Johnson and D. Tjahjono. Does every inspection really need a meeting. *Journal of Empirical Software Engineering*, 3(1):9–35, 1998.

[121] R. Johnson. Documenting frameworks with patterns. In *Proceeding of OOPSLA'92*, pages 63–76, 1992.

[122] R.A. Johnson and G.K. Bhattacharyya. *Statistics: Principles and Methods.* Addison Wesley Publishing Company, Reading, MA., 3rd edition, 1996.

[123] R.E. Johnson and B. Foote. Designing reusable classes. *Journal of Object Oriented Programming*, 1(5):22–35, 1988.

[124] C. Jones. *Patterns of Software Systems Failure and Success.* International Thomson Computer Press, London, UK, 1996.

[125] Y. Kalfoglou and D. Robertson. Applying experienceware to support ontology deployment. In *Proceedings of the 12th International Conference on Software Engineering and Knowledge Engineering (SEKE00)*, 2000.

[126] A. Kamel, S. Voruganti, H. James Hoover, and P.G. Sorenson. Software process improvement model for small organization: An experience report. In *Proceedings of the Annual Oregon Workshop on Software Metrics (AOWEM97)*, 1997.

[127] S. H. Kan. *Metrics and Models in Software Quality Engineering.* Addison Wesley Publishing Company, Reading, MA., 1995.

[128] C. Kaner. The performance of the *n*-fold requirement inspection method. *Requirements Engineering Journal*, 2(2):114–116, 1998.

[129] C. Kaplan, R. Clark, and C. Tang. *Secretes of Software Quality.* McGraw Hill Inc., New York, NY, 1995.

[130] J. Kelly. Inspection and review glossary, Part 1. *The Software Inspection and Review Organization (SIRO) Newsletter*, 2, 1995.
http://www.ics.hawaii.edu/~siro/articles/glossary1.

[131] L. P. W. Kim, C. Sauer, and R. Jeffery. A framework for software development technical reviews. *Software Quality and Productivity: Theory, Practice, Education and Training*, 1995.

[132] B. Kitchenham, A. Kitchenham, and J. Fellows. The effects of inspections on software quality and productivity. Technical Report 1, ICL Technical Journal, 1986.

[133] J. C. Knight and E. A. Myers. Phased inspections and their implementation. *Software Engineering Notes*, 16(3):29–35, 1991.

[134] J. C. Knight and E. A. Myers. An improved inspection technique. *Communications of the ACM*, 36(11):51–61, 1993.

[135] A. Koennecker, R. Jeffery, and G. Low. Implementing an experience factory based on existing organisational knowledge. In *Proceedings of the Australian Software Engineering Conference (ASWEC00)*, 2000.

[136] J. Kolodner. *Case-Based Reasoning*. Morgan Kaufmann Publishers, Inc., 1993.

[137] G.E. Krasner and S.T. Pope. A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, 1988.

[138] S. Kusumoto. *Quantitative Evaluation of Software Reviews and Testing Processes*. PhD thesis, Faculty of the Engineering Science of Osake University, 1993.

[139] S. Kusumoto, A. Chimura, T. Kikuno, K. Matsumoto, and Y. Mohri. A promising approach to two-person software review in an educational environment. *Journal of Systems and Software*, 40:115–123, 1998.

[140] O. Laitenberger. A survey of software inspection technologies. In *Handbook on Software Engineering and Knowledge Management*, volume 2. 2001.

[141] O. Laitenberger, C. Atkinson, M. Schlich, and K. El Emam. An experimental comparison of reading techniques for defect detection in UML design documents. *Journal of Systems and Software*, 53(2), 2000.

[142] O. Laitenberger, K. El Emam, and T. Harbich. An internally replicated quasi experimental comparison of checklist and perspective based reading of code documents. *IEEE Transactions on Software Engineering*, 27(5):387–421, 2000.

[143] R. Lajoie and R. Keller. Design and reuse in object-oriented frameworks, patterns, contracts and motifs in concert. In *Proceedings of the 62 Congress of the Association Canadienne Francaise pour l'Advancement des Sciences*, Montreal, Canada, 1994.

[144] C. Lewis, P. Polson, C. Wharton, and J. Rieman. Testing a walkthrough methodology for theory-based design of walk-up-and-use interfaces. In *Human Factor in Computing Systems, CHI'90 Conference Proceedings*, pages 235–242, Seattle, WA, 1990. ACM Press.

[145] M. Lindvall, I. Rus, and S. Sinha, editors. *proceedings of the 4th International Workshop on Learning Software Organization LSO'01*, Kaiserslauten, Germany, 2002. Fraunhofer Institute for Experimental Software Engineering, Kaiserslautern, Germany.

[146] F. Macdonald. Assist v1.1 user manual. Technical Report RR-96-199 [EFoCS-22-96], Empirical Foundations of Computer Science, (EFoCS), University of Strathclyde, UK, 1997.

[147] F. Macdonald. *Computer Supported Software Inspection.* PhD thesis, Department of Computer Science, University of Strathclyde, 1998.

[148] F. Macdonald and J. Miller. Modelling software inspection methods for the application of tool support. Technical Report RR-95-196 [EFoCS-16-95], Empirical Foundations of Computer Science, (EFoCS), University of Strathclyde, UK., 1995.

[149] J. M. MacLeod. Implementing and sustaining a software inspection program in an r&d environment. *Hewlett-Packard Journal*, 1993.

[150] R. Madachy, L. Little, and S. Fan. Analysis of a successful inspection program. In *Proceedings of the 18th Annual NASA Software Eng. Laboratory Work-shop*, pages 176–198, 1993.

[151] M. Maher. *Cost Accounting: Creating Value for Management.* McGraw Hill Inc., New York, NY, Fifth edition, 1997.

[152] D. Malone. Knowledge management: A model for organizational learning. *International Journal of Accounting Information Systems*, 2002.

[153] S. March and G. Smith. Design and natural science research on information technology. *Decision Support Systems*, 15:251–266, 1995.

[154] M. Marcheli and G. Succi, editors. *eXtreme Programming and Flexible Processes in Software Engineering - XP2000*, Sardinia, Italy, 2000.

[155] J. Martin and W.T. Tsai. *n*-fold inspection: A requirements analysis technique. *Communications of the ACM*, 33(2):225–232, 1990.

[156] V. Mashayekhi, C. Feulner, and J. Riedl. CAIS: Collaborative Asynchronous Inspection of Software. *Software Engineering Notes*, 19(5):21–34, 1994.

[157] Check Mate. Static analysis tools. http://www.bluestone-sw.com/index.html.

[158] S. McConnell. *Code Complete*, chapter 24. Microsoft Press, Redmond, WA, 1993.

[159] S. McConnell. *Rapid Development.* Microsoft Press, Redmond, WA, 1996.

[160] J. McKissick, M.J. Somers, and W. Marsh. Software design inspection for preliminary design. In *Proceedings of the International Computer Software and Applications Conference, COMPSAC'84*, pages 273–281, Las Vegas, NV, Jul 1984. IEEE Computer Society Press, Los Alamitos, Ca.

[161] Merriam-Webster. *Merriam-Webster Dictionary.* http://www.m-w.com/home.htm.

[162] R.E. Merwin. Software management: We must find a way. *IEEE Transactions on Software Engineering*, 1978.

[163] J. Miller. Estimating the number of remaining defects after inspection. *Software Testing, Verification and Reliability*, 9:167–189, 1999.

[164] J. Miller, M. Wood, and M. Roper. Further experiences with scenarios and checklists. *Journal of Empirical Software Engineering*, 3(3):37–64., 1998.

[165] H.D. Mills. Software development. *IEEE Transactions on Software Engineering*, 1976.

[166] B. Minto. *The Pyramid Principle - Logic in Writing and Thinking.* FT Pitman, London, 4th edition edition, 1995.

[167] D. Moody. Using knowledge management and the internet to support evidence based practice. In *Proceedings of the 10th Australian Conference on Information Systems*, pages 660–676, Wellington, New Zealand, 1999.

[168] G. J. Myers. A controlled experiment in program testing and code walkthroughs/inspections. *Communications of the ACM*, 21(9):760–768, 1978.

[169] W. Myers. Shuttle code achieves very low error rate. *IEEE Software*, 5(5):93–95, 1988.

[170] B. A. Nardi. *A Small Matter of Programming.* MIT Press, Cambridge, MA, 1993.

[171] M. Nick and K-D. Althoff. The challenge of supporting repository-based continuous learning with systematic evaluation and maintenance. In *22nd International Conference on Software Engineering. Workshop on Intelligent Software Engineering (WISE3)*, 2000.

[172] I. Nonaka and H. Takeuchi. *The Knowledge Creating Company.* Oxford University Press, 1995.

[173] J.F. Nunamaker, M. Chen, and T.D.M. Purdin. System development in information systems research. *Management Information Systems*, 7(3):89–106, 1991.

[174] E. Ostertag. *A Classification System for software Reuse.* PhD thesis, University of Maryland, 1992.

[175] D. L. Parnas and D. Weiss. Active design reviews: Principles and practices. In *Proceedings of the 8th International Conference on Software Engineering (ICSE8)*, pages 132–136, 1985. Also Available as NRL Report 8927, 18 November 1985.

[176] D. L. Parnas and D. Weiss. Active design reviews: Principles and practice. *Journal of Systems and Software*, 7:259–265, 1987.

[177] *The PEF Model.* A booklet from the PERFECT ESPRIT project 9090 handbook edition, 1996.

[178] J. Perpich, D. Perry, A. Porter, L. Votta, and M. Wade. Anywhere, anytime code inspections: Using the web to remove inspection bottlenecks in large-scale software development. In *Proceedings of the 19th International Conference on Software Engineering (ICSE19)*, pages 14–21, 1997.

[179] P. Polson, C. Lewis, J. Rieman, and C.Wharton. Cognitive walkthrough: A method for theory-based evaluation of user interfaces. *International Journal of Man-Machine Studies*, 36:741–773, 1992.

[180] A. Porter, H. Siy, and L. Votta. A review of software inspections. *Advances in Computers*, 42(4):39–76, 1996.

[181] A. Porter, H.P. Siy, C.A. Toman, and L.G. Votta. An experiment to assess the cost-benefits of code inspections in large scale software development. *IEEE Transactions on Software Engineering*, 23(6):329–346, 1997.

[182] A. A. Porter and L. G. Votta. What makes inspections work? *IEEE Software*, pages 99–102, 1997.

[183] A. A. Porter, L. G. Votta, and V. R. Basili. Comparing detection methods for software requirements inspections: A replicated experiment. *IEEE Transactions on Software Engineering*, 21(6):563–575, 1995.

[184] A.A. Porter and L. Votta. Comparing detection methods for software requirements inspection: A replication using professional subjects. *Journal of Empirical Software Engineering*, 3(4):355–378, 1998.

[185] W. Pree. *Design patterns for object-oriented software development.* Addison Wesley Publishing Company, Reading, MA., 1995.

[186] Taligent Press. *The Power of Frameworks for Windows and OS/2 Developers.* Addison Wesley Publishing Company, Reading, MA., 1995.

[187] R. Prieto-Daz. Implementing faceted classification for software reuse. *Communications of the ACM*, 34(5):89–97, 1991.

[188] J. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1990.

[189] Remote inspection services. http://www.nolan.com/~pnolan/rem_insp.html.

[190] H. Remus. Integrated software validation in the view of inspections/reviews. *Software Validation*, pages 57–65, 1984.

[191] ReviewPro. The automated technical reviews and inspections system. http://www.sdtcorp.com/reviewpr.htm.

[192] S. Rifkin and L. Deimel. Applying program comprehension techniques to improve software inspection. In *Proceedings of the 19th Annual NASA Software Engineering Laboratory Workshop.* NASA, 1994.

[193] D. Rombach. Special presentation in ICSE 2001. In *Proceedings of the 23th International Conference on Software Engineering (ICSE23)*, Toronto, Canada, 2001. ACM Press.

[194] J. Rothfeder. Its late, costly, incompetent but try firing a computer system. *Business Week*, November 7 1993.

[195] G. Ruhe. Learning software organizations. In S.K. Chang, editor, *Handbook of Software Engineering and Knowledge Engineering*, volume vol 1 - Fundamentals. World Scientific Publishing Company, Singapore, 2001.

[196] I. Rus, M. Lindvall, and S.S. Sinha. A state of the art report; Knowledge management in software engineering. Technical report, DoD Data & Analysis Center for Software (DACS), Rome, NY, Dec 2001.

[197] G. W. Russell. Experience with inspection in ultralarge-scale developments. *IEEE Software*, 8(1):25–31, 1991.

[198] K. Sandahl, O. Blomkvist, J. Karlsson, C. Krysander, M. Lindvall, and N. Ohlsson. An extended replication of an experiment for assessing methods for software requirements inspection. *Empirical Software Engineering, An International Journal*, 3(4):327–354, 1998.

[199] G.M. Schneider, J. Martin, and W.T. Tsai. An experimental study of fault detection in user requirements documents. *ACM Transactions on Software Engineering and Methodology*, 1(2):188–204, 1992.

[200] K. Schneider. Realistic and unrealistic expectations about experience exploitation. In *Proceedings of the Conference on Engineering in Software Technology (CONQUEST 2001)*, pages 171–182, Nuernberg, Germany, 2001.

[201] G. Schulmeyer. *Zero Defect Software*. McGraw Hill Inc., New York, NY, 1990.

[202] C. Seaman, M. Mendona, V.R. Basili, and Y.M. Kim. An experience management system for a software consulting organization. In *Proceedings of the 24th Annual Software Engineering Workshop, NASA/GSF*, Software Engineering Laboratory Series, Greenbelt, Maryland, 1999.

[203] C. B. Seaman and V. R. Basili. Communication and organization: An empirical study of discussion in inspection meetings. *IEEE Transactions on Software Engineering*, 24(6):559–572, 1998.

[204] J. Segal. Organisational learning and software process improvement: A case study. In K-D. Althoff, R.L Feldmann, and W. Muller, editors, *proceedings of the 3rd International Workshop on Learning Software Organization LSO'01*, LNCS 2176, pages 68–82. Springer-Verlag, Berlin, Heidelberg, 2001.

[205] Ch.C. Sekar and E.W. Deming. On a method of estimating birth and death rates and the extent of registration. *Journal of the American Statistical Association*, 44(245-248):101–115, 1949.

[206] Y.S. Sherif. The characteristics of efficient formal review. *Microelectronics and Reliability*, 32(3):415–422, 1992.

[207] Y.S. Sherif. Software safety analysis: The characteristics of efficient technical walkthrough. *Microelectronics and Reliability*, 32(3):407–414, 1992.

[208] G. C. Shirey. How inspections fail. In *Proceedings of the 9th International Conference on Testing Computer Software*, pages 151–159, 1992.

[209] B. Shneiderman. Dynamic queries for visual information seeking. *IEEE Software*, 11(6):70–77, 1994.

[210] F. Shull, F. Lanubile, and V.R. Basili. Investigating reading techniques for object-oriented framework learning. *IEEE Transactions on Software Engineering*, 26(11):1101–1118, 2000.

[211] J.L. Simon. *Resampling: The new statistics*. Wadsworth, Boston, 1993.

[212] E. Simoudis. Using Case-Based retrieval for customer technical support. *IEEE Expert*, 7(5):7–13, 1992.

[213] H.P. Siy. *Identifying the Mechanisms Driving Code Inspection Costs and Benefits*. PhD thesis, University of Maryland at College Park, 1996.

[214] B. Smyth and P. Cunningham. A comparison of incremental case-based reasoning and inductive learning. In *Proceedings of the 2nd European Workshop on Case-Based Reasoning*, pages 32–39, 1995.

[215] Software formal inspection guidebook., 1993. http://satc.gsfc.nasa.gov/fi/fipage.html.

[216] Software formal inspection standard. NASA-STD-2202-93, 1993. http://satc.gsfc.nasa.gov/fi/fipage.html.

[217] P. Sparaco. Board faults Ariane-5 software. *Aviation Week and Space Technology*, 145(5):33–34, 1996.

[218] J. Stasko, A. Badre, and C. Lewis. Do algorithm animations assist learning? An empirical study and analysis. In *Human Factor in Computing Systems, INTERCHI'93 Conference Proceedings*, pages 61–66, The Netherlands, 1993. ACM/IFIP, ACM Press.

[219] S. H. Strauss and R. G. Ebenau. *Software Inspection Process*. Systems Design & Implementation Series. McGraw Hill Inc., New York, NY, 1994.

[220] K. E. Sveiby. *The New Organizational Wealth*. Berrett-Koehler Publisher Inc., 1997.

[221] S. Tan, H.-H Teo, B. Tan, and K.-K Wei. Developing a preliminary framework for knowledge management in organizations. In *Proceedings of Fourth Annual Americas Conference on Information Systems*, pages 629–631, Baltimore, MD, 1998.

[222] C. Tautz, K-D Althoff, and M. Nick. A case-based reasoning approach for managing qualitative experience. In *Intelligent Lessons Learned Systems: Papers from the Workshop at 17th National Conference on AI (AAAI00)*., pages 54–59. The AAAI Press, 2000.

[223] C. Tautz and C. Gresse von Wangenheim. REFSENO: A representation formalism for software engineering ontologies. In *Proceedings of the 5th German Conference on Knowledge Based Systems (XPS99), Workshop on Knowledge Management Organization Memory and Reuse*, pages 61–71, 1999.

[224] I. Tervonen. Support for quality-based design and inspection. *IEEE Software*, 13(1):44–54, 1996.

[225] D. Tjahjono. *Exploring the effectiveness of formal technical review factor with CSRS, a collaborative software review system.* PhD thesis, Department of Information and Computer Science, University of Hawaii, 1996.

[226] R. Trittmann. The organic and the mechanistic form of managing knowledge in software development. In K-D. Althoff, R.L Feldmann, and W. Muller, editors, *proceedings of the 3rd International Workshop on Learning Software Organization LSO'01*, LNCS 2176, pages 22–36. Springer-Verlag, Berlin, Heidelberg, 2001.

[227] M.H. van Edman. Structured inspections of code. *Software Testing, Verification and Reliability*, 2(3):133–153, 1992.

[228] G. van Heijst, R. van der Spek, and E. Kruizinga. Organizing corporate memories. In *Proceedings of the 10th Knowledge Acquisition for Knowledge-based Systems Workshop*, Banff, Canada, 1996.

[229] J. Vlissides. *Unidraw Tutorial I: A simple drawing editor.* Stanford University, 1991.

[230] L. G. Votta. Does every inspection need a meeting. *Software Engineering Notes*, 18(5):107–114, 1993.

[231] I. Watson. Case-Based Reasoning tools: An overview. In *Proceedings of the Second UK Workshop on Case Based Reasoning*, pages 71–88, 1996.

[232] G.M. Weinberg. *The Psychology of Computer Programming.* Van Nostrand Reinholt Co, N.Y., 1971.

[233] E. F. Weller. Experiences with inspections at bull hn information system. In *Proceedings of the 4th Annual Software Quality Workshop*, 1992.

[234] E. F. Weller. Lessons from three years of inspection data. *IEEE Software*, 10(5):38–45, 1993.

[235] G. Wenneson. Quality assurance software inspections at NASA Ames: Metrics for feedback and modification. In *Proceedings of the 10th Annual Software Engineering Workshop*, Goddard Space Flight Center, Greenbelt, MD, 1985.

[236] C. Wharton, J. Rieman, C. Lewis, and P. Polson. The cognitive walkthrough method: A practitioner's guide. In J. Nielsen and R.L. Mack, editors, *Usability Inspection Methods*, chapter 5. John Wiley & Sons Inc., New York, NY, 1994.

[237] D. A. Wheeler, B. Brykczinski, and R. N. Jr. Meeson. *Software Inspection - An Industrial Best Practice.* IEEE Computer Society Press, Los Alamitos, Ca., 1996.

[238] D. A. Wheeler, B. Brykczynski, and R. N. Jr. Meeson. Peer reviews similar to inspection. In D. A. Wheeler, B. Brykczinski, and R. N. Jr. Meeson, editors, *Software Inspection - An Industrial Best Practice.*, pages 228–236. IEEE Computer Society Press, Los Alamitos, Ca., 1996.

[239] G. White, D.Anderson, K. Burnham, and D. Otis. Capture recapture method for sampling closed populations. Technical Report LA-8787-NERP, Los Alamos National Laboratory, 1982.

[240] K. Wiig. Comprehensive knowledge management - working paper. Knowledge Research Institute, http://www.knowledgeresearch.com/downloads/compreh_km.pdf, 1999.

[241] A.L. Wolf and D.S. Rosenblum. A study in software process data capture and analysis. In *Proceedings of the Second International Conference on the Software Process*, pages 115–124, 1993.

[242] Q. Yang, E. Kim, and K. Racine. CASEADVISOR: Supporting interactive problem solving and case base maintenance for help desk applications. In *IJCAI'97, Workshop on Practical Use of CBR*, Nogoya, Japan, 1997.

[243] E. Yourdon. *Structured Walkthroughs.* Prentice Hall, New York, NY., 4th edition, 1989.

[244] M.V. Zelkowitz and D.R. Wallace. Experimental models for validating technology. *IEEE Computer*, 31(5):23–31, 1998.

Appendix A

Case Based Reasoning

Case based reasoning [136] is a commonly used knowledge deployment technique. A case-based reasoning system draws a decision on the comparison between knowledge stored in the case base and the new situation. The current problem description and the stored solutions are known as *cases*. The intuition is “what has been done before to successfully solve a problem may be successfully used in similar situations”.

Typically, CBR systems reason using large chunks of knowledge, rules and similarity metrics for adaptation. The CBR technology is an interactive paradigm, where the user is involved in much of the process [231]. The CBR technique is more effective than rule-based or model-based reasoning systems as it can overcome the ”knowledge acquisition bottleneck” by storing cases, as they emerge, for later analysis rather than encoding the entire domain knowledge *a priori* [136]. This appendix provides an introduction to the CBR technique.

A.1 The CBR process

The CBR process can be described as a four-steps cycle [1] - see figure (A.1):

Retrieve: The process starts by a user querying the CBR system to solve a given problem. The system interprets a query to retrieve the most appropriate case(s) from the case base.

Reuse: Often, the system retrieves a list of cases. The user is given a chance

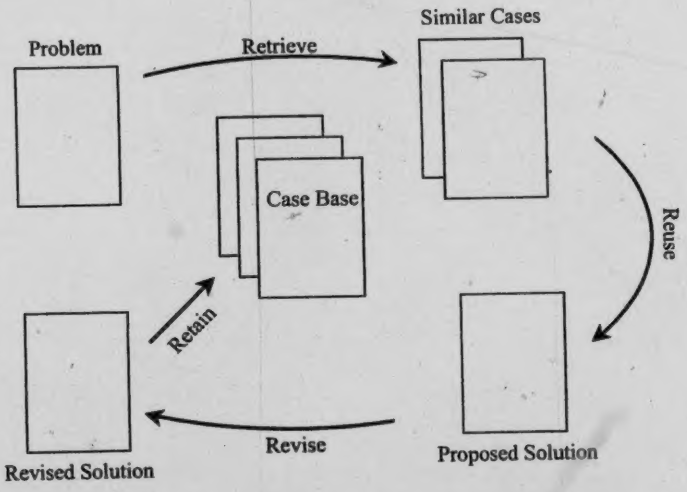


Figure A.1: The process of case based reasoning

to select the 'most' appropriate case from the retrieved list.

Revise: If the selected case does not appropriately solve the current problem, the system (or the user) revises the case to fit the current problem.

Retain If the solution is revised, the system or the user extends the case base to include the revised solution as a new case in the case base.

In reality, most of the available CBR systems are retrieval and reuse systems [231]. Typically, each of the retrieved cases is given a relative score based on their similarity to the problem as expressed in a user querry. Generally, the case with the highest score is the one which the system believes is the most appropriate to answer the user query. Along with the list of cases, the system posts a set of questions that the user can answer. If the user is unsatisfied with the resulting list of cases (e.g. all retrieved cases have similar score), she can answer the questions to further focus the results. Upon answering each question the list of cases and the score of the cases are updated to reflect the answers.

Feature	Value
Make	HP
Processor	P-IV 2.0 GHz
Memory Size	512 MB
Price	$1,200

Table A.1: An example of a case

A.2 Building a CBR System

Typically CBR systems provide case indexing mechanisms in support of retrieval algorithms. The domain expert is usually responsible for defining the specifics of the case base. In this section we will briefly review the main issues related to building a case base and the different techniques CBR designers use to index and retrieve cases.

A.2.1 The case base

The case base is typically the first step in developing a CBR system. A case base consists of a number of cases, each of which describes an experience from the knowledge domain. There are no standards describing what information should be retained in a case [231]; however, each case must encompass a problem description and a particular solution to that problem. Typically, the problem description details the situation in which the case occurred, and the contents of the solution are affected by that situation description.

In general, cases are structured as a collection of feature-value (also called attribute-value) pairs. The main difference among case representations is the granularity of the feature-value pairs. They can be very specific as the example in Table A.1, or very general. In the general type of cases, the case may only have two features, *Case Name* and *Case Solutions*, each is associated with a paragraph of text. Two main factors affect the decision of the case representation: ease of acquisition and provided functionality [136].

A.2.2 Indexing techniques

The objective of indexing the cases is to provide efficient case retrieval. Watson [231] stated some guidelines for efficient indices. Essentially, the indices should be: (1) predictive, (2) allow for an extending case base, and (3) concrete enough to be recognized in the future. Traditionally, indices are used as pointers to cases. Strategies such as applying importance value to cases or sections of cases and labelling cases with their important features have been used to index cases.

A.2.3 Retrieval algorithms

Given a user query, the retrieval algorithms are responsible for retrieving the most similar case in the case base. Several techniques have been used for case retrieval including nearest neighbor, induction and template retrieval or a combination of these strategies [214] [231].

The nearest neighbor approach calculates similarity by matching a weighted sum of the case features. To use this approach, each feature must be assigned a weight to indicate its importance within the knowledge domain. In the inductive approach a decision tree is built to classify cases within the case base. Each branch represents a feature value pair and each level of the tree represents a feature. The most similar cases are retrieved by traversing the tree [188]. Template retrieval algorithms work similarly to the SQL (Standard Query Language) used to extract information from databases. They return all cases that fall within a specified parameters.

A.2.4 Adaptation strategy

There are two main methods to adapt retrieved cases to the current problems [231]: transformational reuse and derivational reuse. These methods depend heavily on the application domain and are expensive to implement. As a result very few CBR systems use adaptation [231].

Appendix B

Modelling Current Peer Reviews

This appendix provides a summary of different review methodologies we encountered in our survey. At a high level of abstraction, reviews are classified as inspections, technical reviews and walkthroughs [108]. Inspections are the most formal process with the most precise objectives and walkthroughs are the least formal that can accommodate a wide range of objectives in one session. Generally, inspections are used for defect detection and elimination, technical reviews are used for building consensus and walkthroughs are used for training [84].

B.1 Inspection

Software inspection, as defined in IEEE STD.610.12-1990 [113] is *"a formal evaluation technique in which software requirements, design or code are examined in detail by person or group other than the author to detect faults, violations of development standards and other problems."*. The inspection objectives as identified in IEEE-STD 1208-1990 [112] is *to detect and identify software elements defects.*

B.1.1 Fagan Inspection

Fagan [77, 78] published an influential method, called inspection, for statically testing a work product to verify that it meets its requirements. There are six principle stages in Fagen's inspection process [78]: planning, overview, preparation, examination, rework and follow-up.

1. *Planning.* A moderator is designated and the inspection team is formed. The moderator ensures that the work product satisfies the inspection entry criteria and assign roles to participants.
2. *Overview.* An optional step to familiarize participants with the work product, its context, etc.
3. *Preparation.* Participants individually, analyze the work product and related materials and record potential defects.
4. *Examination.* The inspection team meets to analyze the work product with the sole objective of finding defects. During the meeting, a person designated as the reader presents the work product while everyone is looking for defects. Identified defects are reported, classified and their severity are noted.
5. *Rework.* The author resolves all the issues noted on the examination stage.
6. *Follow-up.* The moderator verifies that all issues have been resolved effectively and no secondary defects have been introduced, or a second inspection may be scheduled.

A typical inspection team consists of three participants, other than the work product author, drawn from the project technical team; more participants may be added depending on the subject work product. The examination meeting requires several roles to be filled: a *moderator* to orchestrate the meeting, a *reader* to paraphrase the work product and a *maintenance expert* to view the work product from the maintenance perspective.

Only work in progress goes through an inspection process. The work product hast to meet the inspection entry criteria, as well. During the process, supplementary materials should be accessible to inspectors, e.g. checklists, defect distributions from previous inspections, etc. After the inspection, the moderator submits a cumulative report of defects found, their count, type and severity to the project manager.

B.1.2 Fine-tunes on Fagen Inspection

"IEEE Standard for Software Reviews and Audits" [112] (IEEE STD 1028-1988) adopted Fagen's inspection, at large, and fine tuned some of its parameters. Specifically, they restricted the inspection team to a maximum of six participants, and added a 'recorder' role to record the location and description of all defects discovered during the meeting. They also recommended that the preparation period to be within 1.5 hours per inspector.

NASA [216] also fine tuned Fagan inspection focusing primarily on the process itself. The examination meeting is strictly limited to two hours, if the work product examination is not complete, an inspection continuation meeting is scheduled for later time. Additional, informal meetings may be scheduled to resolve open issues raised during the meeting and discuss defect solutions. These meetings are scheduled upon a request from the work product author.

Schneider et al. [199] proposed replicating the inspection process to operate in parallel using N independent teams along with a single moderator who is responsible for coordinating and merging their efforts. They recommended the replication only be applied to the inspection of the user requirements. Studies [155] have showed that traditional inspections are much less successful at detecting requirement faults in this case, than design and code faults.

B.1.3 Gilb Inspection

Gilb [92] introduced an inspection process that, in addition to the main objective of detecting and eliminating defects, included process improvement as a secondary objective. He suggested that inspection should take place between software production phases to ensure a defect free transition between phases.

Gilb [91][92] based his inspection on Fagan's, yet he modified almost all of the stages to adapt for the secondary objective, and introduced a new stage, called third-hour as well. As opposed to Fagan's inspection, the overview step is usually held. It familiarize inspectors with the inspection process as well as the work product to be inspected. Process changes, strategies and productivity goals are usually discussed in this meeting.

During the examination meeting, identified issues are logged. Logged issues can be potential defects, 'question of intent' to the author or an improvement suggestion. Third-hour meeting directly follows the inspection meeting. It is a process brainstorming meeting to discuss causes for raised issues, recommendations for eliminating them in the future as well as improvement suggestions for the inspection process. The rework stage, usually done by the author, resolves the issues logged during the examination meeting. The inspection ends when the moderator make sure that corrective actions to identified defects are taken.

B.1.4 Phased Inspections

A phased inspection [133] [134] is a series of small inspections, termed phases, each of which is designed to inspect one class of defects. The idea behind phased inspection is to examine work products, using a dependable method, against different desired characteristics such as correctness, portability, reusability and maintainability.

There is no overview stage in phased inspections and defect collection is performed individually. Two types of phases exist in phased inspections, single-inspector phase and multiple-inspector phase. The assumption is that all inspectors, in multiple-inspector phase, will find exactly the same list of defects. If not, a meeting may be needed to reconciliate the list of defects. Depending on the property checked, either type of phases will be used. For example, checking design functionality should be performed in a multiple-inspector phase, whereas, checking source code readability may be performed as a single-inspector phase. Inspected work products have to pass phases sequentially, i.e. inspections does not progress to the following phase until all identified defects in the previous phase are reworked.

B.1.5 Inspecting for Program Correctness

Britcher's [42] approach to software review builds on the questionnaire idea used in active design review [176] and Fagan's inspection [77]. Moreover, this approach emphasizes the search for correctness, instead of looking for defects. Reviewers would investigate how the software is developed, informally apply formal verification methods, looking for evidence of disciplined methods in its construction and adequate consideration of the error domain.

Correctness arguments are based on four key program attributes: topology, algebra, invariance and robustness. Topological correctness refers to the hierarchical decomposition into small, manageable and independent subproblems while conserving the original problem space. Algebraic correctness refers to the functional equivalence among successive refitments of the design. Invariance attribute explores the relationship between variables before, during and after execution. Robustness investigates how well the design considers error conditions.

The process involves two to three reviewer in addition to the author. There are two stages in the process, preparation and meeting. The meeting can be split into four sessions [180], each session examines one correctness aspect of the work product.

B.2 Technical Review

A technical review as defined in ANSI/IEEE Std 610.12-1990 [113] is *"a formal meeting at which the preliminary or detailed design of a system is presented to the user, customer or other interested parties for comments and approval"*. Reviews are frequently used to develop consensus about the work product, examine alternatives or identify defects.

B.2.1 Round-Robin Review

The basic idea behind Round-Robin reviews [84] is to give participants an equal and similar share of the entire task. This could be achieved by circulating different review roles among participants or by using forms of redundant roundrobins. Task division could depend on the review criteria, i.e. each person examines the work product against one item of the review checklist, or on the work product itself, i.e. dividing it into sections or on a functional basis. Round-Robin reviews can be used to check work product characteristics or decide the best alternative in design. They also provide a good educational environment, especially when all participants are at the same level of expertise.

The Round-Robin review process is informal to the choice of number of participants, their roles, review results reporting, reviewed material size and maturity level depend on the review objective. For example, for testing work product readability, speed reviewing [165] can be employed. In this technique, the work product is divided into equal parts. Reviewers spend short time (e.g. five minutes) checking a part, before circulating it to the next reviewer. First impression comments provide a good indication for a document readability.

B.2.2 Active Design Review

Active design review, a review process developed by Parnas and Weiss [176], uses a questionnaire to guide the reviewers during their preparation for the review. These questions are carefully designed such that they can only be answered by careful study of the design. The objective of this questionnaires is to enforce the reviewers to take a more active role than just reading the document; some of the questions may ask reviewers to implement particular parts of the design.

There are three stages in active design reviews: overview, examination and follow-up. In the first stage, a brief overview of the module being reviewed is presented. In the second stage, reviewers study the design individually and complete the questionnaire. Reviewers can meet with designers to resolve questions they have about the design and/or the questionnaire. This stage ends by handing the completed questionnaires back to the designers. In the third stage, designers read the completed questionnaires and meet with reviewers to resolve questions the designers may have about reviewers' answers.

B.2.3 Verification Based Reviews

Dyer [70] [71] proposed this review method with the intent of striking a compromise between formality and thoroughness. In these reviews, the work product, usually code, is reviewed line by line using informal correctness proofs.

van Edman [227] has also proposed a review process, mainly for code, that is a cross between formal verification and technical reviews. The method is based on including comments on the code using a formal notation. Then, during the examination meeting, reviewers determine if the comments are adequate and if the code performs the functionality documented in the comments. To successfully apply this review mechanism, another programming methodology, namely the assertion-based methodology, is introduced. Van Edman states that this methodology not only facilitates the review process, but eases the coding process as well.

B.2.4 Selected Aspect Review

'Selected aspect review'[1] is a method of rapidly evaluating material by focusing attention to a few selected aspects, one at a time [84]. This type of review is commonly used as part of feasibility studies, or to reevaluate plans and cost estimate. Large amounts of materials are generally covered, as only selected aspects of selected samples of the work are examined.

In this approach, preparation is recommended but not required and follow-up is the responsibility of the work product author. Before the examination phase starts, each participant should be informed of the primary area of concern, generally by a checklist of items to look for. The roles include moderator, author and reviewers. The moderator role emphasizes on the interpersonal skills more than technical skills because of the review structure. The review team is generally larger in size than other types of reviews.

---

[1] Freeman and Weinberg [84] called this approach inspection, however to avoid confusion, we have used the name selected aspect reviews.

B.2.5 Meeting-less reviews

Several studies [230] [183] have indicated that most defects are found during the preparation stage. Votta [230] suggested replacing the examination meeting with a series of *depositions*. A deposition is a meeting between the work product author, inspection moderator and an inspector to collect his/her findings.

Along the same line, Bisant and Lyle [34] proposed a two person inspection for projects that do not involve large group of developers. The approach requires only two people, the author and a reviewer. In other approaches [238] [158], the process is limited to a person checking the work product and returning his/her comments to the author. These approaches can be limited in participant numbers from one person examines the code in isolation to two or more reviewers examining distributed parts of the work product. Reviewers may meet with the work product author to discuss the defects.

B.3 Walkthrough

A walkthrough is [113] *"a review process in which a designer or programmer leads one or more other members of the development team through a segment of design or code that he or she has written, while the other members ask questions and make comments about technique, style, possible errors, violation of development standards and other problems"*. The primary objective of the walkthrough as identified in ANSI/IEEE-Std-1028-1988 [112] is *"to find defects, omissions, and contradictions, to improve the software element and to consider alternative implementations"*. Other objectives identified on the same standard include *"exchange of techniques and style variations, and education of the participants. A walkthrough may point out efficiency and readability problems in the code, modularity in the design or unstable design specifications."* Software engineers have used walkthroughs for other objectives such as software safety analysis [99], usability analysis [144] [179] and programming language design assessment to determine how easy or hard it is to write a program given a specific language definition [28].

B.3.1 Structured Walkthrough

Yourdon [243] defines his walkthrough process, named structured walkthrough, as *"a peer group review of a product"*, with a basic purpose of error detection. Reviewers are encouraged to make constructive criticism, comments and suggestions about the work product. The spectrum of work products that can undergo a structured walkthrough includes specification, design, code and test documents.

Roles in a structured walkthrough include the presenter (typically the author of the work product), coordinator, secretary to record comments, maintenance oracle to look into maintainability aspects, standards bearer, user representative (where appropriate) and general reviewers to give general comments about the work-product correctness and quality. The roles can be combined, except for the coordinator and secretary. Two persons are a minimum requirement for a walkthrough. Yourdon recommends a team of 5-6 persons for a productive walkthrough. The desired team size depends on parameters such as walkthrough context and level of formality.

A structured walkthrough has two phases, preparation and meeting. Each reviewer should spend on average one hour in preparation, for a typical 30 to 60 minute walkthrough meeting. The meeting begins by reviewing "old business" from previous walkthroughs, if applicable. It may be followed by a brief presentation of the work product. After collecting comments from reviewers, the meeting ends with a recommendation if another walkthrough is needed or not.

A structured walkthrough can be scheduled at different points in the life cycle of a product. Yourdon suggests that they be held on legible documents before any extensive testing, for example, code walkthroughs can be held after clean compilation (legible code) but before testing.

B.3.2 Technical Walkthrough

The technical walkthrough process [206], defined for software safety analysis purposes, is by far the most formally defined walkthrough process. Sherif

[207] recommended technical walkthroughs as a periodical ongoing activity in all software development phases. In addition to the general objective of finding errors, the specific objectives of a technical walkthrough will depend on when it is held in the software life cycle.

The software manager appoints a lead reviewer for each phase of the product life cycle, who in turn, appoints the review team for that phase. Depending on the work product being reviewed, the review team will have three to six participants in addition to the lead reviewer and the author of the work product. For example, the participants for the test plan walkthrough in the analysis phase are: lead reviewer, test-plan author, a system engineer, a senior operations specialist, an integration specialist and a senior development engineer.

Only work-in-progress work-products should undergo a technical walkthrough. The presenter has to distribute the work-product to be reviewed accompanied with a brief statement describing its state of completeness. Participants submit their comments to the lead reviewer before the walkthrough meeting as well. During the meeting, the team goes through the concerns raised and levies action items for those concerns. Critical concerns are added to the project's 'Critical Issues List'. After the meeting, the lead reviewer has to report findings and recommendations to the software manager.

B.3.3 Freedman and Weinberg's Walkthrough

Freedman and Weinberg [84] defined their walkthrough as a review process *"Characterized by the producer of the reviewed material"*. The process is performed on the basis of a step-by-step simulation of procedures. Despite this procedural approach, the walkthrough can be extended to review nonprocedural materials as well.

The objective of this type of walkthrough is mainly educational. Major oversights can also be detected, depending on the background and skills of the audience. There are two roles in this type of walkthrough: presenter, who has to lead the walkthrough meeting as well, and the audience. In this type of walkthrough a classroom approach is followed. This gives the audience a more passive role during the walkthrough meeting and implies that the presenter has to do most of the preparation and guide the meeting progress. The classroom approach is flexible enough to accommodate a large size of audience and large amounts of materials.

B.3.4 Cognitive Walkthrough

Cognitive walkthrough [236] is *"A set of reasonable speculations about a user's background and state of mind while carrying out a task"* that is aimed at the analysis of highly interactive user interfaces. C. Lewis et. al [144] [179] defined this process to evaluate the ease of learning of user interfaces by exploration. Different variations of a cognitive walkthrough have been used in different contexts [28] [218]. The key idea is to examine a plausible sequence of steps leading from a problem to its solution using a tool.

There are two phases in cognitive walkthroughs: preparatory and analysis. In the preparatory phase, reviewers identify the walkthrough inputs. In addition to the design under review, the inputs should cover three different areas: identification of the users, sample task suite for evaluation and action sequences for completing the tasks. During the second phase, the analysis, reviewers examine the actions required to complete each task on the task suite, trying to assess if the user will choose the correct action to complete the task.

There is no limits on the team size of a cognitive walkthrough. It can be as little as one person evaluating his/her own design, up to any number needed to perform the task. Each member in the team should have a specified role. The roles include presenter (the designer), secretary, coordinator and reviewers (analysts). The reviewers should contribute various expertise such as knowledge of potential market, user-needs analysis, and interface design evaluation. A cognitive walkthrough can be performed at different points in the development of a user interface.

B.3.5 Programming Walkthrough

A programming walkthrough [28] applies the same basic idea of cognitive walkthrough to assess the ease of writing programs in a programming language.

The two processes share the property of defining specific tasks. The reviewers must have a suite of problems to examine the process of writing programs to solve them. In contrast to the cognitive approach, programming walkthroughs do not require the action sequences for completing the program to be given a *priori*. The output of the walkthrough is the careful documentation of these sequences, along with the knowledge required for each step in the sequence. Language designers use this knowledge to understand the programming process from the programmer's point of view.

Appendix C

Support Material for the Case Study

This appendix provides a copy of forms and tools we developed and used during our case study. The items listed in this appendix are:

1. The questionnaire used to survey the students' feedback about the peer review process. This questionnaire was added after the first round of projects. It was added to solicit the the students' opinion about the review right after it was enacted.
2. The questionnaire used to solicit students' feedback about the framework and its documentation as well as the review processes recommended during development.
3. The checklist used in the study as a part of the review preparation package. In the first round of projects, two checklists were used, one copied from Freedman and Weinberg book [84], and one developed to along the lines of the scenario based reviews. The second checklist is included in Section C.3.
4. Student preparation form.
5. Review meeting collection form.
6. Student's background form.

It should be noted that the students' participation in the review process was optional. All those attending the review meetings agreed to release the review results to be used in this research project. Students' consent was captured on video before the review meeting.

C.1 Post-review questionnaire

1. After the CSF review, how confident are you that you will be able to use the framework to produce an application? (Answer on a scale of 1 to 5, where 1 = not confident and 5 = very confident)

2. Rate the relevance and clarity of the documentation in terms of gaining a high level understanding the CSF framework. In addition what percentage of time did you spend on each of the types of documentation? (For relevance and clarity, answer on a scale of 1 to 5, where 1 = not useful/clear and 5 = very useful/clear. For time, simply give a percentage.)

Item	Relevance/Usefulness	Clarity	Time
Code			
Hooks			
Examples			
Design diagrams			
Use cases			
Guidelines/Process			
FAQ			
Talking to the CSF developer			

3. How did you approach preparation for the review?
   For example - started by looking over the review guidelines, reading the use cases, then tried an example, etc.

4. Did you try any examples? If so, which ones?

5. Characterize the changes to the communication and persistence requirements of your application due to using the CSF? (Answer either uncertain, or using a scale of 1 to 5, where 1 = no significant changes, 5 = significantly modified)

6. How helpful was the review process for improving your understanding of the framework? (Answer on a scale of 1 to 5, where 1 = not helpful and 5 = very helpful)

7. From your perspective, how can the review be improved for CMPUT 401 students in the future?

C.2 Post-project questionnaire

Section I

I.1. What is your name and group name?

I.2. For your part of the project, how much did you use the CSF?
(Please answer: significant use, limited use or did not use)

If you used the CSF significantly or in a limited way, please answer sections II and III. If you were not involved in using the CSF, proceed to section III.

Section II

II.1 How did you approach development using the CSF? Give a brief description. For example, did you start by modifying examples, read the documentation, examine the code, etc.?

II.2 Now that you've completed a project, rate the usefulness/relevance and clarity of the following on a scale of 1 to 5 for using the CSF. (1 = not useful or clear, 5 = very useful or clear)

Item	Relevance/Usefulness	Clarity
Code		
Hooks		
Examples		
Design diagrams		
Use cases		
Guidelines/Process		
FAQ[1]		
Talking to the CSF developer		

II.3 Which option or combination of options from question 2.2 did you rely on in learning and using the CSF?

II.4 What problems did you encounter in using the CSF? Give a brief description of 3 to 5 of the major ones.

II.5 At each stage of development, how much time did you personally spend on issues involving the CSF as opposed to other activities? (i.e. How much time did the CSF take up for that stage only as compared to other activities at that stage?) Answer on a scale of 1 to 5 where:

1 = up to 10% (a minimal amount of your time for that stage)

2 = 10 to 25%

3 = 26 to 33%

4 = 34 to 50%

5 = greater than 50% (the majority of your time for that stage)

* Project planning
* Analysis
* Design
* Implementation
* Testing

II.6 On a scale of 1 to 5, how difficult did you find each of the following parts of the framework to use? (1 = very difficult, 5 = very easy) Leave it blank if you didn't use that part.

* Asynchronous communication
* Synchronous communication
* Data master/data proxy
* Persistence
* Mail servers
* The Framework as a whole

II.7 After completing the project, how confident are you that you will be able to use the framework to produce another application? (Answer on a scale of 1 to 5, where 1 = not confident and 5 = very confident)

Section III

III.1 How useful was the design review in the areas of:
(Answer on a scale of 1 to 5 where 1 = not useful and 5 = very useful)

- Using the CSF correctly
- Enhancing the overall design quality
- Enhancing the design documentation style and details
- Enforcing a milestone for the progress of the project

III.2 How appropriate did you find the timing of the reviews?
(Answer: too early in the process, too late in the process, or at an appropriate time.)

- CSF review
- Design review

C.3 Scenario-based checklist

This objective of this checklist is to help you as a reviewer to:

- Identify the parts of the CSF external interface that are relevant to your particular application.
- Identify missing information (hooks, data objects, etc.) that is needed by your application.

In order to use this checklist you have to:

i) Read the CSF documentation.

ii) Visualize your application's requirements' in the areas covered by the CSF.

1. Identify the set of CSF use cases that are relevant to your application.
2. Identify the set of CSF Hooks that you will use to develop your application.
3. For the previously identified hooks and use cases:
    - Do they cover your application's key functional requirements? If not, identify the functionalities that may not be properly covered.
4. Identify the CSF's classes that you need to inherit from/ modify/ etc. for your application's purposes.
5. Identify the CSF's input/output data objects relevant to your application.
6. For the previously identified classes and objects:
    - **a.** Do they cover your application's anticipated requirements, if not, identify the requirement(s) that may not be covered.
    - **b.** Does the CSF documentation cover their functionality, intended use, etc. If not, identify any information need to be added.
7. Define key collaborations between the CSF and your application. I.e. in the existing collaboration diagrams:
    - **a.** Identify those relevant to your application.
    - **b.** For each of these diagrams, mark the objects that your application might change.

C.4 Forms

COMPUT 401 – Software Engineering
CSF Review Preparation Form.

Team: --
Reviewer: --
Preparation Time: --------------------------------

Findings:

Type	Severity	Document	Comment
-------	----------	-------------	--
-------	----------	-------------	--
-------	----------	-------------	--
-------	----------	-------------	--
-------	----------	-------------	--
-------	----------	-------------	--
-------	----------	-------------	--
-------	----------	-------------	--
-------	----------	-------------	--
-------	----------	-------------	--
-------	----------	-------------	--
-------	----------	-------------	--
-------	----------	-------------	--
-------	----------	-------------	--
-------	----------	-------------	--
-------	----------	-------------	--
-------	----------	-------------	--
-------	----------	-------------	--
-------	----------	-------------	--
-------	----------	-------------	--
-------	----------	-------------	--
-------	----------	-------------	--
-------	----------	-------------	--
-------	----------	-------------	--
-------	----------	-------------	--
-------	----------	-------------	--
-------	----------	-------------	--
-------	----------	-------------	--

Notes:

- Record the preparation time to the nearest 10 minutes.
- Classify the type of finding into either (question or concern)
- Classify the severity into: major, moderate or minor.
- For 'document' indicate what part of the documentation is related to your finding.
- **In comment explain the actual finding.**

COMPUT 401 – Software Engineering
Review Collection Form.

Review Time: Date:
Team Name:
Moderator:
Recorder:
Reviewers:

Participants:

Findings:

Type	Severity	Document	Comment

COMPUT 401 – Software Engineering
Background Information.

Personal Information:

Name: --
ID: --

Educational Background:

Current Degree: -- Year: ----------------
Pervious Degree(s) (if any): --
Please list all computing courses completed (course ID and name):

1. -- 2. --
3. -- 4. --
5. -- 6. --
7. -- 8. --
9. -- 10. --

Hands-on Experience:

Please describe any software projects you have worked on. Briefly explain the project size (total number of software workers, and total project duration in weeks) and your role in the project (design, coding, testing, documentation, etc.). Also indicate if it is a course project or not.

Project Size		Your Role	Project Description
No. of Workers	Duration		
-------	-------	-----------	--
-------	-------	-----------	--
-------	-------	-----------	--
-------	-------	-----------	--

//steele-lk/docs/stdbkgrd.doc Student Background – version 1.0 1 of 1